Exact real arithmetic in computer algebra

Hur, Namhyun

Award date:
2001

Awarding institution:
University of Bath

Exact Real Arithmetic in Computer Algebra

submitted by

Namhyun Hur

for the degree of Doctor of Philosophy

of the

University of Bath

2001

ProQuest LLC
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106-1346

Acknowledgements

I cannot thank enough my supervisor Prof. James H. Davenport. James jump-started my study when everything seemed to have stalled. I am deeply grateful for his guidance, help and encouragement throughout the study. I would also like to thank James for providing me with a part-time post as a research officer for the Esprit OpenMath project.

Many thanks to the Department of Mathematics for the scholarship without which I would not be able to complete this PhD.

A special thanks to my grandmother for her love, to my parents, for bringing me to this world, to my parents-in-law and brothers-in-law for their support, to my wife, Eunkyung, for her unending love and support, to my daughter, Seungyeon, for her big hugs when I needed them.

Also I would like to thank my friends at the Bath Computing Group: Robin, Nick, Bill, Natee, Adam, Joy Dave. Thanks also goes to Gabrielle, Mary and Parade friends, Kim, Claire, Bruce, and anyone else I have forgotten.

Summary

In this thesis we study exact real arithmetic in terms of computer algebraic viewpoint. Floating point arithmetic is not exact and the problems caused by this inexactness is well-known. Exact real arithmetic is an effort to overcome this difficulty of floating point arithmetic and can generate guaranteed approximations of real numbers up to any given precision.

But exact real arithmetic has its own problems. Although exact, it still has the fundamental problem of equality: we cannot, in general, decide whether two real numbers are equal or not. This thesis describes an effort to solve this equality problem for the special case of real algebraic numbers.

We start by describing a history of exact real arithmetic and then discuss its representations and implementations. We implement the lazy B-adic representation in **Axiom**. Using this implementation of exact real arithmetic, we derive a symbolic-numeric representation of real algebraic numbers and implement it in Axiom. Using the exactness of the numerical part, we can devise an equality algorithm for the real algebraic numbers and with a little more effort we can also get a generic root operation for the real algebraic numbers. The concept of real root separation is at the heart of these algorithms. We also compare performances within variants of the lazy B-adic model.

Glossary

$\mathbb{N}$ the set of natural numbers

$\mathbb{Q}$ the set of rational numbers

$\mathbb{R}$ the set of real numbers

$\mathbb{R}_{pr}$ the set of primitive recursive real numbers

$\mathbb{R}_c$ the set of computable (or recursive) real numbers

$\mathbb{R}_{ldr}$ the lazy dyadic representation of $\mathbb{R}_c$

$\mathbb{R}_{lft}$ the linear fractional representation of $\mathbb{R}_c$

$\mathbb{R}_{pair}$ a model for real algebraic numbers

$\mathbb{R}_{LDRPair}$ a model for real algebraic numbers with numerical part $\mathbb{R}_{ldr}$

Contents

Acknowledgements 1

Summary 2

Glossary 3

Table of Contents 4

1 Introduction 7
- 1.1 Exact Real Arithmetic . . . 7
- 1.2 What this thesis is about . . . 8
- 1.3 Organization of Thesis . . . 8
- 1.4 Logical Terminology . . . 9

2 Literature Survey 11
- 2.1 Introduction . . . 11
- 2.2 Non-Exact Computing with Real Numbers . . . 12
- 2.3 Exact Computing with Real Numbers . . . 14
  - 2.3.1 Computable Real Numbers: $\mathbb{R}_c$ . . . 14
  - 2.3.2 $\mathbb{R}_c$ as a field . . . 15
- 2.4 The Undecidability of Zero . . . 17

3 Lazy Dyadic Exact Arithmetic 19
- 3.1 Introduction . . . 19
- 3.2 Rounding to the nearest: `ndiv` . . . 20
- 3.3 Making incremental: `memo` . . . 20
- 3.4 Separating reals from zero: `separate` . . . 21
- 3.5 `exp(x)` in terms of Taylor series . . . 22
  - 3.5.1 Taylor formula with the remainders . . . 23
  - 3.5.2 `exp` for $|\mathtt{x}| \leq 1$ . . . 23

4 Linear Fractional Exact Real Arithmetic ($\mathbb{R}_{lft}$) 27
- 4.1 Introduction . . . 27
- 4.2 Gosper's Continued Fraction Arithmetic . . . 27
  - 4.2.1 linear fractional form (`lff`) . . . 28
  - 4.2.2 inputing a digit into an `lff`: $x \mapsto q + \frac{1}{x}$ . . . 28

4.2.3 outputing a digit from an lff: $x \mapsto x - \frac{1}{q}$ . . . 28
4.2.4 binary operations on lff . . . 29
4.3 Vuillemin's Exact Real Arithmetic . . . 30
4.3.1 a computable rounding . . . 30
4.3.2 $\mathbb{R}_c$ as infinite products of homographies . . . 32
4.3.3 Algebraic Algorithm . . . 32
4.3.4 homographic algorithm . . . 34
4.3.5 quadratic algorithm . . . 34
4.3.6 transcendental functions . . . 35
4.4 Potts and Edalat's Exact Real Arithemtic: $\mathbb{R}_{lft}$ . . . 35
4.4.1 $\mathbb{R}_c$ in $\mathbb{R}_{lft}$ . . . 36
4.4.2 functions in $\mathbb{R}_{lft}$ . . . 38
4.5 Heckmann's Work on $\mathbb{R}_{lft}$ . . . 38

5 A Category for $\mathbb{R}_c$ in Axiom **40**
5.1 Introduction . . . 40
5.2 LazyOrderedField: a Category for $\mathbb{R}_c$ . . . 41
5.2.1 lazy set . . . 43
5.2.2 lazy set with one operation . . . 44
5.2.3 lazy set with two operations . . . 46
5.2.4 lazy ordering . . . 48
5.2.5 lazy ordered field . . . 49
5.3 LazyReal as an Extension of LazyOrderedField . . . 50
5.4 LDR ($\mathbb{R}_{ldr}$) as a domain of LazyReal . . . 50
5.4.1 LDR as a domain of LazyReal . . . 50
5.5 Pair: a Category for $\mathbb{R}_{pair}$ . . . 51
5.6 LDRPair as a domain of Pair . . . 52
5.6.1 LDRPair as a domain of Pair . . . 52

6 An Exact Algebraic Arithmetic **53**
6.1 Introduction . . . 53
6.2 LDRpair: an Exact Representation of $\mathbb{R}_a$ . . . 54
6.3 LDRpair Arithmetic . . . 54
6.4 An Equality for LDRpair . . . 55
6.4.1 Example 1 : $\sqrt{2} \times \sqrt{3} = \sqrt{6}$ . . . 57
6.4.2 Example 2 : $\sqrt{9 + 4\sqrt{2}} = 1 + 2\sqrt{2}$ . . . 58
6.5 An Inequality for LDRpair . . . 59
6.6 A generic rootOf Operation for LDRpair . . . 59
6.6.1 Finite Product of Simple Radicals . . . 60
6.6.2 Finite Summation of Simple Radicals . . . 61

7 Implementations in Axiom **63**
7.1 Introduction . . . 63
7.2 $\mathbb{R}_{ldr}$ as a domain of LazyReal . . . 63
7.2.1 basic arithmetic operations . . . 63

7.2.2 ordering . . . 65
7.2.3 transcendental functions . . . 66
7.3 $\mathbb{R}_{lft}$ as a domain of `LazyReal` . . . 67
7.4 $\mathbb{R}_{LDRpair}$ as a domain of `Pair` . . . 69
7.4.1 arithmetic operations . . . 69
7.4.2 the `refine` operation . . . 70

8 Performance Comparisons **71**
8.1 Introduction . . . 71
8.2 Comparisons within `LDR` . . . 71
8.2.1 finite summation . . . 71
8.2.2 finite product . . . 72
8.3 Comparisons among `LDR`, `LQR` and `LHR` . . . 73
8.3.1 finite summation . . . 73
8.3.2 finite product . . . 73
8.3.3 $\sqrt{\sqrt{\sqrt{\sqrt{x}}}}$ . . . 73

9 Conclusion and Further Study **75**
9.1 Overview of thesis . . . 75
9.2 Conclusion . . . 77
9.3 Further Research Topics . . . 77

A Proofs of $\mathbb{R}_{ldr}$ algorithms **79**

B log, sin **and** arctan **in** $\mathbb{R}_{ldr}$ **84**
B.1 $\texttt{ln}\frac{(\texttt{1+x})}{(\texttt{1-x})}$ for $|\texttt{x}| < \frac{1}{2}$ . . . 84
B.2 `sin(x)` for $\texttt{0} \leq \texttt{x} \leq \pi/4$ . . . 86
B.3 `arctan(x)` for $|\texttt{x}| \leq \texttt{1}$ . . . 88

Bibliography **89**

Chapter 1

Introduction

1.1 Exact Real Arithmetic

Exact real arithmetic is an arithmetic developed from the desire to compute real numbers exactly as much as we can. Hence the term exact real arithmetic can be misleading since we cannot calculate real numbers exactly in general. **Lazy exact real arithmetic** might be a better term in that sense.

The theory of exact real arithmetic is based on the concept of computable real numbers. Roughly speaking, a computable real number is a real number which we can represent in a sufficiently large computer. For a rigorous definition of computable real numbers, see [4]. Since every computer is finite we can easily find out that the number of computer-representable real numbers are finite although there are infinitely, but countably, many computable real numbers. The theory of recursion is an important part of the area called mathematical logic and the so called Turing-Church thesis asserts that the concept of recursion and computability are equivalent.

There are many ways we can construct the real number system. Among these the Cauchy sequence approach and the Dedekind cut approach are the mostly known. The Cauchy sequence approach seems to fit better with recursion since recursion is inherently sequential. Indeed one of the exact real representation, called a *finite b-adic model* [7, 34], is a mixture of Cauchy sequence and recursive sequence. Although not impossible, it looks very difficult to devise an exact real arithmetic model by mixing the Dedekind cut and resursion since the notion of recursive set seems harder to realize. We can also construct real numbers based on their numerical expansion (see [10] for an example of a construction of real numbers based on binary expansion) and thus develop exact real arithmetic based on this. The diagram below is a rough summary of these combinations.

$$\begin{aligned} \text{Cauchy sequence} + \text{Recursion} &\Rightarrow \text{Finite } b\text{-adic model} \\ \text{Numerical Expansion} + \text{Recursion} &\Rightarrow ? \\ \text{Dedekind cut} + \text{Recursion} &\Rightarrow ? \end{aligned}$$

Another well-known model called a *linear fractional transformation exact real arithemtic*

[38] is based on the continued fraction expansion of real numbers.

Why is exact real arithmetic important? Well, we can answer this question in two aspects: practical and theoretical (mathematical). In practice, exact real arithmetic gives you the absolute confidence that the result of a particular computation is correct, up to a given precision, whereas floating point arithmetic cannot. Hence it can be used as an error checker for floating point computation. Knowing that what you have just calculated is safe from errors makes you happy. Theoretically or mathematically, exact real arithmetic can be useful in answering some of the difficult questions such as the normality of the Euler number e[1] in addition to its obvious use to answer to the kind of question "What is the n-th digit of a real number x?".

But exact real arithmetic has its own problems. One of the most serious problems is the inability to decide equality of two numbers. This is not suprising since no amount of checking that finite segments of two numbers are the same can prove that they are exactly the same. Currently there seems to be of no way to resolve this problem in general although we will have a say about this for the special subset of real algebraic numbers later in this thesis. Another problem of exact real arithmetic is its inefficiency compared to floating point arithmetic. Exact real arithmetic tends to be a lot slower than floating point arithmetic although, as far as we know, no precise comparison has been done so far.

1.2 What this thesis is about

This thesis is about exact real arithmetic. More specifically it is about the implementation and development of exact real arithmetic in a computer algebra system. We focus on the lazy exact real arithmetic both for description and implementation and we only give description for the linear fractional exact real arithmetic. For the computer algebra system we chose Axiom[2]. We could implement them in any language or in any computer algebra system but it was a deliverate choice for the reason that we want to realize computable real numbers as an abstract data type inside the Axiom's powerful algebraic structure. Axiom looks ideal for this task.

1.3 Organization of Thesis

In Chapter 2, we give a brief survey on exact computing with real numbers. We start with problems of floating point arithmetic, focusing on the issue of precision. Then, we describe computing real numbers exactly. We describe the fundamental concept of *computable real numbers*, its history, properties and problems. We end this chapter by highlighting the problem of zero undecidability.

In the next two chapters we describe the two main models of exact real arithmetic. In Chapter 3, we describe the lazy dyadic exact real arithmetic (base 2) [7, 34]. This

[1] A real number is called normal with respect to a given base b if all the digits have equal asymptotic frequencey of occurrences in its b-adic expansion. It is not known whether e is normal or not

[2] Axiom is a powerful computer algebra system, originally developed by IBM with the name of Scratchpad, but now licensed by NAG.

model represents a computable real number by a recursive Cauchy sequence which converges to the real number. We describe some key properties of this model such as non-incrementality. Several key operations are described: `ndiv`, a rounding integer division to keep the error down to 1/2, `memo`, a caching to keep the evalation incremental, and `separate`, an operation for separating two computable real numbers. This chapter ends with a description of the exponential function in terms of its Taylor series. In Chapter 4 we describe the linear fractional model [38]. It is based on Gosper's fundamental work on continued fraction which is then further developed by Vuillemin. We describe Gosper's seminar work on continued fraction arithmetic, Vuillemin's exact real arithmetic, and Potts and Edalat's linear fractional exact real arithmetic. We end with a brief mention of Heckmann's work on linear fractional model.

In Chapter 5, we specify and implement computable real numbers as a category in Axiom. Algebraically speaking, the set of computable real numbers is a field but without equality. This forces us to build a category for such fields. We then define the lazy dyadic model and the linear fractional model as domains of the category. We also define a category for real algebriac numbers and define exact models of them as domains of the category.

In Chapter 6, we describe an exact real algebraic arithmetic. Real algebraic numbers are exactly represented by a symbolic-and-numeric representation. We define its arithmetic. Using this repesentation we show that we can solve equality testing problem, one of the big disadvantage of numeric-only approaches to exact real arithmetic. As a by-porduct we also have a generic `rootOf` operation.

In Chapter 7, we describe implementations in Axiom.

In Chapter 8, we compare the performances. First we compare the evaluation performances of several expressions within the lazy dyadic model. Then we compare the performances of the lazy dyadic model with other variants, the lazy quartic model (base 4) and the lazy hexadecimal model (base 16).

In the conclusion we summarize the whole thesis, points out further study area.

1.4 Logical Terminology

Here we define several logical concepts which will be used mainly in the next chapter. We followed the defintions given in [2].

Definition 1.4.1 (bounded formula) *A formula is called bounded if all its quantifiers are bounded.*

Definition 1.4.2 (definability of a set $A \in \mathbb{N}^k$ in $\mathbb{N}$) *A set $A \in \mathbb{N}^k$ is called definable in $\mathbb{N}$ by a bounded formula if there exist a function $f : \mathbb{N}^k \to \mathbb{N}$ and a bounded formula F such that*

$$(n_1, \ldots, n_k) \in A \quad \textit{if and only if} \quad \mathbb{N} \models F[n_1, \ldots, n_k]$$

Definition 1.4.3 (primitive recursive set) *A set $A \in \mathbb{N}^k$ is called primitive recursive if it is definable in $\mathbb{N}$ by a bounded formula.*

However, the case of functions is different. There are functions definable in $\mathbb{N}$ by a bounded formula which are not primitive recursive, for example, the Ackermann's function.

Definition 1.4.4 (definability of a function $f : \mathbb{N}^k \to \mathbb{N}$ in $\mathbb{N}$) *A function $f : \mathbb{N}^k \to \mathbb{N}$ is called definable in $\mathbb{N}$ by a bounded formula if there exists a bounded formula H such that*

$$f(n_1, \ldots, n_k) = b \text{ if and only if } \mathbb{N} \models H[n_1/x_1, \ldots, n_k/x_k, b/y]$$

Definition 1.4.5 (primitive recursive function: 1) *The class of primitive recursive functions is the smallest class C of functions such that*

1. *All constant functions, $\lambda x_1 x_{2k}.m$, are in C, $\forall k, m.1 \leq k, 0 \leq m$.*
2. *The successor function, $\lambda x.x + 1$ is in C.*
3. *All idnetity functions, $\lambda x_1 x_{2k}.x_i, 1 \leq i \leq k$, are in C.*
4. *If f is a function of k variables in C, and $g_1, g_2, \ldots, g_k$ are (each) functions of m variables in C, then the function $\lambda x_1 \cdots x_m.f(g_1(x_1, \ldots, x_m), \ldots, g_k(x_1, \ldots, x_m))$ is in C.*
5. *If h is a function of $k+1$ variables in C, and g is a function of $k-1$ variables in C, then the unique function f of k variables satisfying*

$$\begin{aligned} f(0, x_2, , x_k) &= g(x_2, , x_k) \\ f(y+1, x_2, , x_k) &= h(y, f(y, x_2, , x_k), x_2, \ldots, x_k) \end{aligned}$$

is in C.

Using the above standard definition of primitive recursive functions, we can derive another characterisatin as below [2].

Theorem 1.4.6 (primitive recursive function: 2) *A function $f : \mathbb{N}^k \to \mathbb{N}$ is called primitive recursive if and only if*

1. *f is definable in $\mathbb{N}$ by a bounded formula*
2. *there exists a primitive recursive function $g : \mathbb{N}^k \to \mathbb{N} \neq f$ such that*

$$\forall (n_1, \ldots, n_k).f(n_1, \ldots, n_k) \leq g(n_1, \ldots, n_k)$$

Using the above characterization of primitive recursive function one can check that most basic number-theoretic functions like the greatest common divisor function, are primitive recursive. Recursive functions can be characterized by simply dropping the second condition from the definition of primitive recursive functions.

Definition 1.4.7 (recursive function) *a function $f : \mathbb{N}^k \to \mathbb{N}$ is called recursive if it is definable in $\mathbb{N}$ by a bounded formula.*

Chapter 2

Literature Survey

2.1 Introduction

[34] points out three main false beliefs in computing with floating point arithemtic. These are:

1. (rounding) rounding-off errors are negligible and rarely occurs only for very ill-conditioned problems,
2. (operation) the less operations, we get less inaccuracy, and
3. (precision) increasing precision, we get more accuracy.

To demonstrate the above problems [34] gave two examples: calculating the limit of the sequence of rational numbers defined by

$$f(n) = \begin{cases} \frac{11}{2} & \text{if } n = 0 \\ \frac{61}{11} & \text{if } n = 1 \\ 111 - \frac{1130 - 3000/f(n-2)}{f(n-1)} & \text{otherwise} \end{cases}$$

and computing the real values of a function defined by

$$f(n) = n! \times \left(e - \sum_{k=0}^{n} \frac{1}{k!} \right).$$

These problems (and many others which we may not aware of) show the need and importance of exact real arithmetic. In this chapter we give a breif survey about exact real arithmetic and it is organised as follows:

- In Section 2.2, we give another example which shows how complicated the issue of precision in floating point arithmetic is.
- In Section 2.3, we describe exact real arithmetic: its history, its algebraic properties (field).
- Finally we describe the most important problem of exact real arithmetic: the undecidabiltiy of zero [40].

we then move to the area of exact real number computation. The central concept of exact real computation is the notion of computable real numbers. A brief history of computable real numbers is given. We also show that they form a field. Finally we end with one of the most important problems of exact real arithmetic: the undecidability of zero.

2.2 Non-Exact Computing with Real Numbers

There are real numbers that simply are not computable. Then the natural question is how we should compute those that are computable. Even today we compute or rather approximate real numbers by using a subset of the rational numbers, called floating point numbers. Hence true arithmetic is performed by limited-precision arithmetic. But the problem of the accumulation of rounding errors in this representation is now well known and often led to disastrous results.

Some people have put forward multiple-precision floating point arithmetic [3] to mend such problems. But we can show that mere growth of precision on floats does not solve the problem. As an example we can try to compute $e^{\pi\sqrt{163}}$. This number is very close to being an integer. Below is an Axiom session for computing the above number using the Axiom's arbitrary-precision floating point system. In Axiom, the default precision is 20 decimal digits which we can check by using the function `precision()`.

```
(1) -> precision()
   (1)  68                          Type: PositiveInteger
```

Evaluating the above number with the default precision gives

```
(2) -> exp1()^(pi()*sqrt(163))@Float
   (2)  26253741 2640768743.97  Type: Float
```

Now let's increase the precision to, say 26 digits, and see what happens.

```
(3) -> precision(precision()+22)
   (3)  68                          Type: PositiveInteger
(4) -> exp1()^(pi()*sqrt(163))@Float
   (4)  26253741 2640768744.0   Type: Float
```

As we can see the number looks like an integer. But we can see that it is not by increasing precision further then evaluate the number again as below.

```
(5) -> precision(precision()+28)
   (5)  90                          Type: PositiveInteger
(6) -> exp1()^(pi()*sqrt(163))@Float
   (6)  26253741 2640768743.9999999999 9925007  Type: Float
```

This example shows how complicated the issue of precision is and therefore why we need to compute real numbers exactly. Another example which shows the complicateness of the precision comes from the real root isolation problem. Consider the polynomial $18x^3 + 4x^2 - 12x + 3$. This polynomial has two positive and one negative real roots:

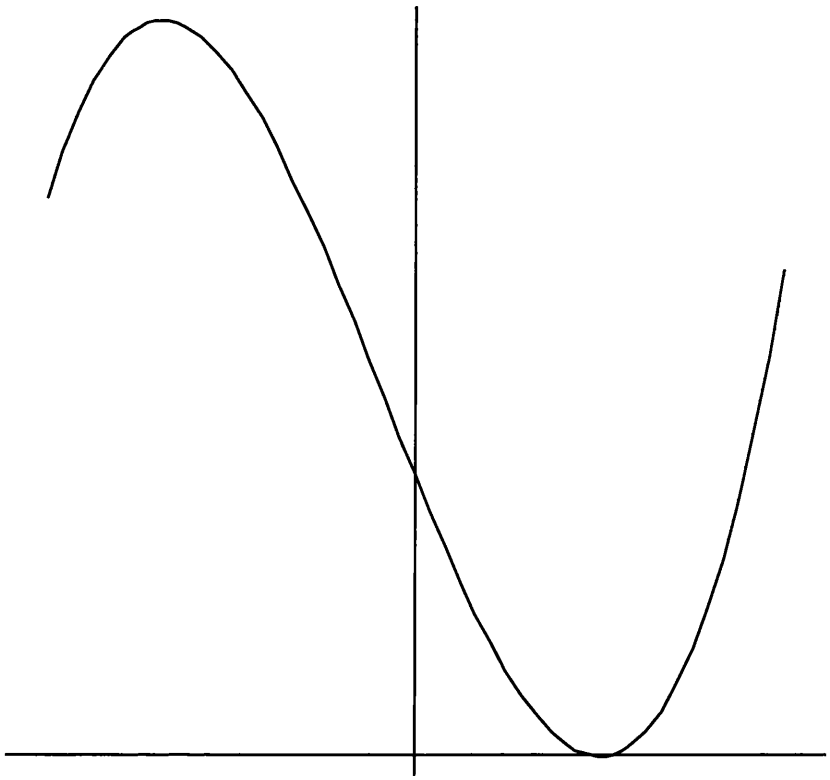


Figure 2.1: This figure shows the two very close roots of $18x^3 + 4x^2 - 12x + 3$

$-1.028, 0.421$, and 0.385. As we can see from the figure 2.1 the two positive are not equal but they are quite close. This example provide a good example of separation bounds which will be discussed in Chapter 6. Another real world example for the seriousness of precision we can refer to the Patriot missile failure during the Gulf War.

2.3 Exact Computing with Real Numbers

Thus recently there has been renewed interest in performing real arithmetic exactly. The first rigorous study of exact real arithmetic has been carried out by Boehm *et al.* [7]. They reported experiments with various representations of computable real numbers and advocated using the functional approach. This approach has been followed by an extensive study by Ménissier-Morain [34]. This model is often called finite B-adic exact real arithmetic, but we will call it *lazy dyadic exact real arithmetic* or, in short, *lazy dyadic real* ($\mathbb{R}_{ldr}$) arithmetic. Another model for exact real arithmetic has been studied by Vuillemin [47]. His model is based on continued fraction arithmetic following the pioneering work by Gosper [15]. One particular feature of this model is that it can deal with infinite quantities. Vuillemin's work is then followed by Potts and Edalat [38] who combined Vuillemin's approach with domain theory. This model is based on linear fractional transformation which is a generalization of continued fraction. So we will call it *linear fractional transformation exact real arithmetic.* But this is too long so we use a shorter name *linear fractional transformation real* ($\mathbb{R}_{lft}$) arithmetic.

2.3.1 Computable Real Numbers: $\mathbb{R}_c$

In this section we deal with the following questions about the computable real numbers ($\mathbb{R}_c$) :

1. Who started all this? (history)
2. What are they? (definition(s))
3. Why do we have to learn them? (usage, importance)
4. What are the problems they have? (undecidability)

We start with a short history about $\mathbb{R}_c$ by particularly mentioning Specker's work on primitive recursive real numbers. Then we gather various characterizations of $\mathbb{R}_c$ followed by interesting comparative remarks. It is very important to distinguish between $\mathbb{R}_c$ and the various models of $\mathbb{R}_c$[1]. Next we show that $\mathbb{R}_{ldr}$, hence $\mathbb{R}_c$, forms a field[2].

A short history of $\mathbb{R}_c$

Cantor showed $\mathbb{R}$ is uncountable using a diagonalization technique. In fact he showed that the set of real numbers in the interval $[0, 1]$ is uncountable. The uncountability of

[1] This will be realized in Chapter 5 where we define $\mathbb{R}_c$ as a *category* in Axiom and models of $\mathbb{R}_c$ as *domains* in that category

[2] $\mathbb{R}_{ldr}$ is also real closed [28].

real numbers implies that there are infinitely many uncomputable real numbers since we can only compute countably many real number with a finite computer. Then Specker [43], as an effort to give a constructive foundation for mathematics, first defined the notion of primitive recursive real numbers ($\mathbb{R}_{pr}$). A real number x is called *primitive recursive* if there is a primitive recursive function $F_x : \mathbb{N} \to \mathbb{Q}$ such that $|F_x(n) - x| < 2^{-n}$. In practice, rather than using $F_x : \mathbb{N} \to Q$ with $|F_x(n) - x| < 2^{-n}$ approximating x, we use $f_x : \mathbb{N} \to Z$ with $|f_x(n) - 2^n x| < 1$. which approximates x with accuracy 2^n for all $n \in \mathbb{N}$. This is the same as saying that

Definition 2.3.1 (primitive recursive real numbers) *A real number x is primitive recursive if it can be approximated by a primitive recursive function $f_x : \mathbb{N} \to Z$ such that*

$$\left| \frac{f_x(n)}{2^n} - x \right| < 2^{-n}$$

or, equivalently

$$|f_x(n) - 2^n x| < 1.$$

Indeed one of the characterizations of the computable real numbers, namely $\mathbb{R}_{ldr}$, is a direct extension of $\mathbb{R}_{pr}$. But $\mathbb{R}_{pr}$ defined by some primitive recursive functions only form a very narrow class of real numbers as shown by Specker [43]. Thus we will miss out lots of real numbers if we represent them with primitive recursive functions.

$\mathbb{R}_c$ forms a much larger class of real numbers than $\mathbb{R}_{pr}$ and they also do not suffer from the same problems as $\mathbb{R}_{pr}$ do [43]. Now we define $\mathbb{R}_c$[3].

Definition 2.3.2 *A real number x is called computable if there is a recursive function $f_x : \mathbb{N} \to Q$ which approximates x with the accuracy 2^n for all $n \in \mathbb{N}$.*

But, in practice, we use a more convenient but equivalent definition than 2.3.2.

Definition 2.3.3 (computable real numbers) *A real number x is computable if it can be approximated by a recursive function $f_x : \mathbb{N} \to Z$ such that*

$$|f_x(n) - 2^n x| < 1.$$

From the definition we can easily see that all the integers and thus rationals are computable since, for any $k \in \mathbb{Z}$, we can define a recursive function $f_k := n \mapsto 2^n k$ and clearly this satisifes the required inequality for any n.

2.3.2 $\mathbb{R}_c$ as a field

We show here that $\mathbb{R}_{ldr}$ forms a field.[4] $\mathbb{R}_{ldr}$ represents an $x \in \mathbb{R}_c$ by a function $\mathbf{x} : \mathbb{N} \mapsto \mathbb{Z}$ such that $|\mathbf{x}(n) - 2^n x| < 1$. Notice that we changed notation for clarity and use

[3]Of course there are other characterizations of $\mathbb{R}_c$: as a limit of recursive Cauchy sequence, as a limit of shrinking nested intervals. For these and proofs of their equivalences see [34]

[4]$\mathbb{R}_{ldr}$ will be the subject of the next chapter (Chapter 3) and thus some of the proofs below could have been given there. [34] shows the same but using a different characterization of $\mathbb{R}_c$: as recursively enumerable sequences of nested intervals

typewrite font $\mathtt{x}$ to denote the $\mathbb{R}_{ldr}$ representation of x. First we show that $\mathbb{R}_{ldr}$ is closed under addition. If we let

$$\mathtt{x}+\mathtt{y} := n \mapsto \left\lfloor \frac{f_x(n+2)+f_y(n+2)}{4} \right\rceil$$

for any $\mathtt{x},\mathtt{y} \in \mathbb{R}_{ldr}$, where $\lfloor\ \rceil$ denotes rounding to the nearest integer, then

Theorem 2.3.4

$$\forall n.\ \ \mathtt{x},\mathtt{y} \in \mathbb{R}_{ldr} \Rightarrow \mathtt{x}+\mathtt{y} \in \mathbb{R}_{ldr}$$

.

Proof We have to show that $|(\mathtt{x}+\mathtt{y})(n) - 2^n(x+y)| < 1$.

$$\begin{aligned}
|(\mathtt{x}+\mathtt{y})(n) - 2^n(x+y)| &= \left| \left\lfloor \frac{\mathtt{x}(n+2)+\mathtt{y}(n+2)}{4} \right\rceil - 2^n(x+y) \right| \\
&\leq \frac{1}{2} + \left| \frac{\mathtt{x}(n+2)+\mathtt{y}(n+2)}{4} - 2^n(x+y) \right| \\
&= \frac{1}{2} + \frac{1}{4}\left| (\mathtt{x}(n+2)+\mathtt{y}(n+2)) - 2^{n+2}(x+y) \right| \\
&\leq \frac{1}{2} + \frac{1}{4}\left| \mathtt{x}(n+2) - 2^{n+2}(x) \right| + \frac{1}{4}\left| \mathtt{y}(n+2) - 2^{n+2})(y) \right| \\
&< \frac{1}{2} + \frac{1}{4} + \frac{1}{4} \\
&= 1.
\end{aligned}$$

Next we show that $\mathbb{R}_{ldr}$ is also closed under multiplication. If we let

$$\mathtt{xy} := n \mapsto \left\lfloor \frac{\mathtt{x}(n+m+2)\mathtt{y}(n+m+2)}{2^{m+2}} \right\rceil$$

where we choose m such that if either $|z-x| < 1$ or $|z-y| < 1$ then $|z| < 2^m$. In other word, choose m so that $2^m > max(x,y)+1$. Then we have

Theorem 2.3.5 $\forall n.\ \ \mathtt{x},\mathtt{y} \in \mathbb{R}_{ldr} \Rightarrow \mathtt{xy} \in \mathbb{R}_{ldr}$.

Proof We have to show that $|\mathtt{xy}(n) - 2^n(xy)| < 1$.

$$\begin{aligned}
|\mathtt{xy}(n) - 2^n(xy)| &= \left| \left\lfloor \frac{\mathtt{x}(n+m+2)\mathtt{y}(n+m+2)}{2^{m+2}} \right\rceil - 2^n(xy) \right| \\
&\leq \frac{1}{2} + \frac{1}{2^{m+2}}\left| \mathtt{x}(n+m+2)\mathtt{y}(n+m+2) - 2^{n+m+2}xy \right| \\
&< \frac{1}{2} + \frac{1}{2^{m+2}}\left| \mathtt{x}(n+m+2) - 2^{n+m+2}x \right| \\
&\quad + \frac{1}{2^{m+2}}\left| \mathtt{y}(n+m+2) - 2^{n+m+2}y \right| \\
&< \frac{1}{2} + \frac{2^m}{2^{m+2}} + \frac{2^m}{2^{m+2}} \\
&= \frac{1}{2} + \frac{1}{4} + \frac{1}{4} \\
&= 1.
\end{aligned}$$

Also every nonzero $\mathbf{x} \in \mathbb{R}_{ldr}$ has a multiplicative inverse. First we choose a positive integer m so that $|\mathbf{x}(n)| > 2^{-m}$ hence $|\mathbf{x}(n)| > 2^{-(m+1)}$. Now let

$$\mathbf{x}^{-1} := n \mapsto \left\lfloor \frac{1}{\mathbf{x}(n+2m+2)} \right\rceil .$$

Then we have

Theorem 2.3.6 [5] $\forall n.\ \mathbf{x} \in \mathbb{R}_{ldr} \Rightarrow \mathbf{x}^{-1} \in \mathbb{R}_{ldr}$.
Proof We have to show that $\left|\mathbf{x}^{-1}(n) - 2^n x^{-1}\right| < 1$.

$$\begin{aligned}
\left|\mathbf{x}^{-1}(n) - 2^n x^{-1}\right| &= \left|\left\lfloor \frac{1}{\mathbf{x}(n+2m+2)} \right\rceil - 2^n x^{-1}\right| \\
&\leq \frac{1}{2} + \left|\frac{1}{\mathbf{x}(n+2m+2)} - 2^n x^{-1}\right| \\
&= \frac{1}{2} + |x|^{-1} \left|\mathbf{x}(n+2m+2)\right|^{-1} 2^{-(2m+2)} \left|\mathbf{x}(n+2m+2) - 2^n x\right| \\
&\leq \frac{1}{2} + \frac{2^m 2^{m+1}}{2^{2m+2}} \left|\mathbf{x}(n+2m+2) - 2^{n+2m+2} x\right| \\
&< \frac{1}{2} + \frac{1}{2} \\
&= 1.
\end{aligned}$$

Thus we have shown that $\mathbb{R}_{ldr}$, hence $\mathbb{R}_c$, forms a field.

2.4 The Undecidability of Zero

In whatever way we compute real numbers there is a fundamental obstruction, the so-called undecidability of zero. Among the consequences of the undecidability of zero are the undecidability of the integer part of a computable real number and the undecidability of rationality. The undecidability of the integer part of a computable real number in turn implies that we can not determine exactly the continued fraction of a computable real number and the b-ary expansion of a computable real number.

Hence the above mentioned models for $\mathbb{R}_c$ suffer from this problem. These models are bad and sometimes impossible at determining zero. But there is a way through this problem assuming Schanuel's conjecture. Richardson [41] proposed to solve this problem using a method which combines a numerical proof of non-zeroness and an algebraic proof of zeroness. We will not describe Richardson's method here but we just note that the method uses sophisticated techniques such as the LLL algorithm [11] and Wu's method [49]. Interested readers should see [41]. Also Richardson's method only solves the problem for elementary numbers, not for all computable numbers. The

[5] More precisely we have to prove $\forall \mathbf{x} \in \mathbb{R}_{ldr}.\mathbf{x} \neq 0 \Rightarrow \exists \mathbf{y} \in \mathbb{R}_{ldr}.\mathbf{xy} = 1$. But checking the hypothesis $\mathbf{x} \neq 0$ is impossible in $\mathbb{R}_{ldr}$ which is the topic of the next section. What we can do instead is that we can check whether $\mathbf{x}$ is *separated* from 0 if $\mathbf{x}$. A real number is called separated from 0 if there exists a rational number r such that $0 < r < |x|$. This important notion of separation forms the basis of ordering relations and will be described in more detail later

elementary numbers are complex numbers which are implicitly or explicitly defined using algebraic operations and exponentiation.

Chapter 3

Lazy Dyadic Exact Arithmetic

3.1 Introduction

$\mathbb{R}_{ldr}$[1], is a model of $\mathbb{R}_c$. It represents a computable real number x by a B-adic[2] sequence $\mathbf{x}$ such that for all n

$$\left|\frac{\mathbf{x}(n)}{2^n} - x\right| < \frac{1}{2^n}.$$

or equivalently

$$|\mathbf{x}(n) - 2^n x| < 1.$$

Thus, in $\mathbb{R}_{ldr}$, an integer k can be represented by a function $\mathbf{k} := n \mapsto 2^n k$ since

$$|\mathbf{k}(n) - 2^n k| = |2^n k - 2^n k| = 0 < 1.$$

Since $\mathbb{R}_{ldr}$ forms a field as we have shown in the previous chapter, one should be able to represent all the usual field operations by such functions in $\mathbb{R}_{ldr}$ (Actually the closedness proofs with respect to addition, multiplication, and inverse in Chapter 2 directly gives algorithms for the corresponding operations. Other proofs are given in Appendix 1). Furthermore one can also represent transcendental functions by several methods such as truncated Taylor series. [34, 33] briefly mentions that this can be done. But [19] gave an explicit calculational proof of the exponential function, represented by a truncated Taylor series, using the theorem prover `HOL`. The proof given in [19] is algorithmic in the sense that we can actually extract an algorithm for the exponential function from the proof. What we describe here is actually a completion of the *algorithm extraction* following the guidance given in [19].

This chapter is organised as follows:

- In Section 3.2, we define an integer division operation, `ndiv`.
- In Section 3.3, we discuss the issue of incrementality and the `memo` function.

[1] The definitive reference on this topic is [34, 33] which provides both algorithms and their correctness proofs for most functions.

[2] We chose $B = 2$, hence the title.

- In Section 3.4, we discuss the notion of separation and the `separate` operation.
- In Section 3.5, we define an exponential function in terms of truncated Taylor series (defintions of log, arctan and sin are given in Appendix 2).

3.2 Rounding to the nearest: `ndiv`

We start with a preliminary remark about an integer division (named as `ndiv` following [18]) operation which will be frequently used. This operation is necessary due to the fact that the truncation error from ordinary integer division can be almost 1 and thus violates the bound condition of the 2-adic model. Thus `ndiv` is an integer division which rounds to the nearest integer so that the error is $\leq 1/2$:

$$\left| l \texttt{ ndiv } p - \frac{l}{p} \right| \leq \frac{1}{2}.$$

Note that the definition of ndiv depends on the rounding of the integer division of the base integer arithmetic. If the integer division, say *div*, rounds towards $-\infty$ (eg. the big integer division in the *bignum* library of Caml) then the definition below

$$l \texttt{ ndiv } p := \begin{cases} l & \text{if } p = 1 \\ (l + p \text{ div } 2) \text{ div } p & \text{otherwise} \end{cases}$$

will do the job. But if *div* rounds toward to 0 (most systems adopt this eg. Axiom) then the definition of `ndiv` must take into account for the $x < 0$ case. In such cases, `ndiv` can be defined as

$$l \texttt{ ndiv } p := \begin{cases} l & \text{if } p = 1 \\ (l + p \text{ div } 2) \text{ div } p & l > 0 \\ (l - p \text{ div } 2) \text{ div } p & \text{otherwise} \end{cases}$$

The `ndiv` contributes an error $\leq \frac{1}{2}$ and this fact will be frequently used in establishing the error bounds later.

As for actual definitions (and proofs of correctness) of many elementary operations such as $+, -, \times, /$ see [18, 34] and Appendix 1.

3.3 Making incremental: `memo`

Unfortunately $\mathbb{R}_{ldr}$ is not incremental. This means that, in $\mathbb{R}_{ldr}$, the calculation of the $(n+k)$-th approximant of $\mathbf{x}$, $\mathbf{x}(n+k)$, does not use the n-th approximant of $\mathbf{x}$, $\mathbf{x}(n)$ as we can see below in the case of addition.

$$\mathbf{x} + \mathbf{y} := n \mapsto \lfloor \frac{\mathbf{x}(n+2) + \mathbf{y}(n+2)}{4} \rceil.$$

The main advantage of incrementality is that the previous result can be reused to increase precision by calculating by some further digits. Fortunately, we can make $\mathbb{R}_{ldr}$

incremental by using the memo theorem below which allows us to cache the most recent approximation.[3]

Theorem 3.3.1 (memo) [4]

$$\forall n, \forall k \geq 1. \quad \left|\mathtt{x}(n+k) - 2^{n+k}x\right| < 1 \quad \Rightarrow \quad \left|\left\lfloor \mathtt{x}(n+k)/2^k \right\rfloor - 2^n x\right| < 1 \qquad (3.1)$$

Using Theorem 3.3.1 we can define a memo function which immediately returns $\mathtt{x}(n+k)$ `ndiv` 2^k. Its implementation in Axiom is an exact copy of that in a functional language (camllight) given in [18].

$$\mathtt{memo(x)} := n \mapsto \begin{cases} \mathrm{r} & \text{if } n \leq i \text{ and } n = i, \text{ where mem} = \mathtt{ref}[-1,0] \\ & [\mathrm{i,r}] = \mathtt{deref}(\mathrm{mem}) \\ \mathtt{ndiv}(\mathrm{r},\ 2^{i-n}) & \text{if } n \leq i \text{ and } n \neq i, \\ (\mathrm{mem} = [\mathrm{n,x\ n}];\ \mathtt{x\ n}) & \text{if } n > i \end{cases}$$

Note that the definition of memo function returns $\mathtt{x}(n)$ from $\mathtt{x}(n-1)$ while the notion of incrementality is about getting $\mathtt{x}(n+k)$ from $\mathtt{x}(n)$. Note that in both definitions, the initial reference assignment must be outside the main definitions.

3.4 Separating reals from zero: separate

Although, in general, we can not test $x = 0$ or not in exact real arithmetic, we can check whether x is separated from 0 or not, or more precisely, semidecidable . A computable real number x is called *separated* from 0 if there exists a rational number r such that $0 < r < |x|$ [36, 18]. Although it might run forever in case $x = 0$, we can define such an operation which recursively separates the given real number from 0. In practice, finding an integer ≥ 1 is enough since the denominators are of fixed form.

$$\mathtt{sepAux\ (x)} \quad := \quad n \mapsto \begin{cases} \mathtt{x}(n) & \text{if } |\mathtt{x}(n)| > 0 \\ \mathtt{sepAux\ x}\ (n+1) & \text{otherwise.} \end{cases}$$

$$\mathtt{sep\ (x)} \quad := \quad \mathtt{sepAux\ x}\ 0.$$

But in order to order two numbers we have to separate two numbers rather than one as below[5].

$$\mathtt{separateAux(x,y)} \quad := \quad n \mapsto \begin{cases} \mathtt{x}(n) - \mathtt{y}(n) & \text{if } |\mathtt{x}(n) - \mathtt{y}(n)| > 1 \\ \mathtt{separateAux\ (x,y)}\ (n+1) & \text{otherwise.} \end{cases}$$

$$\mathtt{separate\ (x,y)} \quad := \quad \mathtt{separateAux\ (x,y)}\ 0.$$

[3] A hybrid incremental reperesentation combining signed digits approach and lazy b-adic approach is proposed in [45].

[4] See Appendix 2 for proof.

[5] This definition and the following orderings are copied from [18]. We could define `sep` using `separate` but then we will have to change the condition part.

Note that we needed the condition $|\mathtt{x}(n)-\mathtt{y}(n)| > 1$ in the defintion of `separateAux`, whereas we needed the condition $|\mathtt{x}(n)| > 0$ in `sepAux`, due to the $1/2$ contribution from each x and y. We can also justify the above condition by a simple numerical counter-example. Let $\mathtt{x}(4) = 17$ and $\mathtt{y}(4) = 18$, i.e., $|\mathtt{x}(4) - \mathtt{y}(4)| = 1$. Then all we can say is that $1 < x < \frac{18}{16}$ (since $|\frac{17}{16} - x| < \frac{1}{16}$) and $\frac{17}{16} < y < \frac{19}{16}$ (since $|\frac{18}{16} - y| < \frac{1}{16}$) but this does not necessarily mean $x < y$. We can define lazy orderings using `separate` [18].

$$\begin{aligned}
\mathtt{x} < \mathtt{y} &:= \mathtt{separate}(\mathtt{x},\mathtt{y}) < 0 \\
\mathtt{x} \leq \mathtt{y} &:= \mathtt{x} < \mathtt{y} \\
\mathtt{x} > \mathtt{y} &:= \mathtt{separate}(\mathtt{x},\mathtt{y}) > 0 \\
\mathtt{x} \geq \mathtt{y} &:= \mathtt{x} > \mathtt{y}
\end{aligned}$$

Note that two different orderings are involved above: lazy dyadic ordering and integer ordering. Using `sep` we can define constructive versions of several functions which involve equality in their classical counterparts. For example, a classical sign function can be defined as

$$sign\ (x) := \begin{cases} 1 & \text{if } x > 0 \\ -1 & \text{if } x < 0 \\ 0 & \text{otherwise.} \end{cases}$$

Using `sep` we can define a constructive version of sign.

$$\mathtt{sign}\ (\mathtt{x}) := \begin{cases} 1 & \text{if } \mathtt{sep}\ \mathtt{x} > 0 \\ -1 & \text{if } \mathtt{sep}\ \mathtt{x} < 0 \\ 0 & \text{otherwise.} \end{cases}$$

Here we will define another ordering, $=_n$, which we will use in Chapter 6.

$$\mathtt{x} =_n \mathtt{y} := m \mapsto \mathtt{x}(m) = \mathtt{y}(m) \quad \forall m \leq n.$$

`sep` finds the numerator k such that $|k/2^n| > 0$. But we can also find the smallest n such that $|k/2^n| > 1$. This function is often called `msd` (most significant digit).

$$\mathtt{msdAux}\ (\mathtt{x}) := n \mapsto \begin{cases} n & \text{if } |\mathtt{x}(n)| > 1 \\ \mathtt{msdAux}\ \mathtt{x}\ (n+1) & \text{otherwise.} \end{cases}$$

$$\mathtt{msd}\ (\mathtt{x}) := \mathtt{msdAux}\ \mathtt{x}\ 0.$$

Note that `msd` is a property of `x`, not x. Different representations of x may have different `msds`.

3.5 exp(x) in terms of Taylor series

There are several methods to represent elementary functions. One of the favorite is the Taylor series which, although not the fastest, is relatively simple and nice. In this section we define `exp` in terms of truncated Taylor series.

The main task in representing elementary functions using truncated Taylor series, whether in floating point arithmetic or exact real arithmetic, is to find the upper bounds of the neglected or remaining terms when we truncate the series at a certain term. The exact upper bounds are usually given in the form of an integral but we can usually find estimates for the upper bounds from them either by tricks or by using the Lagrange version of Taylor's theorem which expresses a function in terms of the partial sums of its Taylor series and the estimation of the remainders without integral [16].

We also have to identify another important error bound, which is special to $\mathbb{R}_{ldr}$ due to the way it models $\mathbb{R}_c$. In summing the truncated terms, each term contributes a certain and guaranteed error bound and we have to specify the accumulation of these *modelling errors.*

We describe here only `exp`. Definitions of other functions, `ln`, `sin` and `arctan` are given in Appendix B. We first state the Lagrange verion of Taylor's theorem and use this to derive the truncation errors. The derivation of modelling errors are exercises in inequality proofs although not too simple.

3.5.1 Taylor formula with the remainders

We will use the following theorem by Lagrange.

Theorem 3.5.1 *Let $f(x)$ be a function continuous on $[a, x]$ and m-times differentiable on (a, x). Then there exists a $\xi \in (a, x)$ such that*

$$f(x) = \sum_{i=0}^{m-1} \frac{(x-a)^i}{i!} f^{(i)}(a) + \frac{(x-a)^m}{m!} f^{(m)}(\xi)$$

where $f^{(m)}$ denotes m-th differentiation.

The advantage of 3.5.1 is that we can estimate the remainder without integration. For example, since e^x is certainly continuous on, say $[0, 2]$, and infinitely many differentiable on $(0, 2)$ with $e^m = e$ for any m, letting $\xi = 1 \in (0, 2)$ we are assured that

$$e^x = \sum_{i=0}^{m-1} \frac{x^i}{i!} + \frac{x^m}{m!} e(1)$$

If we assume $|x| \leq 1$, Theorem 3.5.1 immediately gives us $3/m!$, as an upper bound for the remainders since

$$\left| \frac{x^m}{m!} e(1) \right| < \frac{3}{m!} \quad (\text{for } |x| \leq 1).$$

3.5.2 exp for $|x| \leq 1$

We extract an algorithm for `exp(x)` by closely following the HOL proof for e^x given in [19]. Formula 3.2 tells us we have to find two kinds of error bound: one, the **modelling error** due to the summation term $\sum_{i=0}^{m-1} \frac{\mathtt{x}^i}{i!}$ and the other, **truncation error** due to the term $\frac{\mathtt{x}^m}{m!}\mathtt{e}$ (`e` denotes `exp(1)`). The most important point here is that the overall error must be less than 1.

First, since $|\mathbf{x}| \leq 1$, 3.5.1 gives us immediately an upper bound for the remainders

$$\left|\frac{x^m}{m!}e\right| \quad < \quad 3/m!$$

Note that the above $\leq$ is a lazy ordering.

Next we find the total accumulation error in the summation of m terms. Let us denote the guaranteed error bound for the i-th term, $\mathbf{t}_i = \mathbf{x}^i/i!$, by $\mathbf{k}_i(n)$. In [19] k_i should be $\mathbf{k}_i(n)$ to be precise.

$$\left|\mathbf{t}_i(n) - 2^n\frac{x^i}{i!}\right| \quad \leq \quad \mathbf{k}_i(n)$$

and we calculate the next term $\mathbf{t}_{i+1}$ by using the defintion

$$\mathbf{t}_{i+1} \quad := \quad n \mapsto (\mathbf{x}(n)\mathbf{t}_i(n)) \ \mathbf{ndiv}\ 2^n(i+1)$$

and notice that now we have another source of error due to the **ndiv** division. Then we can derive a formula[6] which expresses $\mathbf{k}_{i+1}(n)$ in terms of $\mathbf{k}_i(n)$

$$\mathbf{k}_{i+1}(n) \quad = \quad \frac{2\mathbf{k}_i(n)}{i+1} + \frac{1}{(i+1)!} + \frac{1}{2}$$

since

$$\begin{aligned}
\left|\mathbf{t}_{i+1}(n) - 2^n\frac{x^{i+1}}{(i+1)!}\right| \quad &\leq \quad \frac{1}{2} + \left|\frac{\mathbf{x}(n)\mathbf{t}_i(n)}{2^n(i+1)} - \frac{\mathbf{x}(n)}{2^n(i+1)}\frac{2^n x^i}{i!}\right| + \\
&\qquad \left|\frac{\mathbf{x}(n)}{2^n(i+1)}\frac{2^n x^i}{i!} - \frac{2^n\mathbf{x}^{i+1}}{(i+1)!}\right| \\
&= \quad \frac{1}{2} + \left|\frac{\mathbf{x}(n)}{2^n(i+1)}\right|\left|\mathbf{t}_i(n) - 2^n\frac{x^i}{i!}\right| + \frac{x^i}{(i+1)!}\left|\mathbf{x}(n) - 2^n x\right| \\
&\leq \quad \frac{1}{2} + \left|\frac{\mathbf{x}(n)}{2^n(i+1)}\right|\mathbf{k}_i + \frac{x^i}{(i+1)!} \\
&= \quad \frac{1}{2} + \frac{\mathbf{k}_i}{i+1}\left(\left|\frac{\mathbf{x}(n)}{2^n} - x\right| + |x|\right) + \frac{x^i}{(i+1)!} \\
&\leq \quad \frac{1}{2} + \frac{2\mathbf{k}_i}{i+1} + \frac{1}{(i+1)!}
\end{aligned}$$

Hence using the above formula we can finally conclude

Theorem 3.5.2

$$\forall i.\ \ \mathbf{k}_i(n) \quad \leq \quad 2$$

[6] Harrison [19] proves this formula using the HOL theorem prover.

Proof We have $\mathtt{k}_0(n) = 0$ since $\mathtt{t}_0 = 1$, $1(n) = 2^n 1$ thus $|2^n 1 - 2^n 1| = 0$. We also have

$$\begin{aligned} \mathtt{k}_{i+1}(n) &= \frac{2\mathtt{k}_i(n)}{i+1} + \frac{1}{(i+1)!} + \frac{1}{2} \\ &= \frac{2i!}{2(i+1)!}\mathtt{k}_i(n) + \frac{2+(i+1)!}{2(i+1)!} \\ &\leq 2 \end{aligned}$$

since $\frac{2i!}{2(i+1)!} < 1$, $\mathtt{k}_i(n) < 1$ (because of guaranteed error bound) and $\frac{2+(i+1)!}{2(i+1)!} < 1$. So the total error in the summation of m terms is always less than or equal to $2m$

$$\left| \sum_{i=0}^{m-1} \frac{\mathtt{x}^i}{i!} \right| \leq 2m$$

since the modelling error,$\mathtt{k}_i$, for each term $\mathtt{x}_i/i!$ is always ≤ 2.

Recall that we have an additional error $\leq 1/2$ which originates from the division in the definition 3.2. Since we have identified all the possible errors we can derive an algorithm for `exp(x)` for $|\mathtt{x}| \leq 1$ to any accuracy. To make the overall error less than 1 as required, and noting that we have `ndiv` division error $\leq 1/2$, we make

- the truncation error, $|\mathtt{exp(x)}(n) - \sum_{i=0}^{m-1} \frac{\mathtt{x}^i}{i!}|$, less than $1/4$ and
- the summation error, $|\sum_{i=0}^{m-1} \mathtt{t}_i|$, less than or equal to $1/4$.

Then the overall error will be strictly less than $1/4 + 1/4 + 1/2 = 1$.

To make the truncation error $< 1/4$ we find m such that

$$\left| \mathtt{exp(x)}(n) - \sum_{i=0}^{m-1} \frac{\mathtt{x}^i}{i!} \right| = \left| \frac{\mathtt{x}^m}{m!}\mathtt{e} \right| < \frac{1}{2^{n+2}}$$

Since

$$\left| \frac{\mathtt{x}^m}{m!}\mathtt{e} \right| < \frac{3}{m!}$$

finding an m such that

$$\frac{3}{m!} < \frac{1}{2^{n+2}}$$

is enough.

To make the summation error $\leq 1/4$, since the error in the summation of m terms is bounded by $2m$ and recalling the finite summation operation[7] we find an l such that

$$2m \leq 2^l$$

and then evaluate each $\mathtt{x}$ (we have m such $\mathtt{x}$s) at $(n + l + 2)$. Then the error, after division by 2^{l+2} will be $\leq 1/4$.

Hence the total error, not forgetting the error from `ndiv` division, will be strictly less than 1 as required. Summarizing all this we have

[7] See Appendix A.

Theorem 3.5.3 *Let*

$$\texttt{exp(x)} := n \mapsto \sum_{i=0}^{m-1} \texttt{t}_i(n) \texttt{ ndiv } 2^{l+2}$$

where

$$\texttt{t}_i := n \mapsto \begin{cases} 2^{n+l+2} & \text{if } i = 0 \\ (\texttt{x}(n)\texttt{t}_{i-1}(n)) \texttt{ ndiv } 2^n(i) & \text{otherwise.} \end{cases}$$

Then

$$|\texttt{exp(x)}(n) - 2^n e^x| \quad < \quad 1.$$

For $|\texttt{x}| > 1$ one can use the relation $e^{2x} = (e^x)^2$ to reduce the range until $|\texttt{x}| \leq 1$.

Chapter 4

Linear Fractional Exact Real Arithmetic ($\mathbb{R}_{lft}$)

4.1 Introduction

The linear fractional transformation (`lft`) [38] used in $x \in \mathbb{R}_{lft}$ is a generalization of the continued fraction. Khinchin [27] found the task of arithmetic with continued fraction almost impossible. But this problem was solved by Gosper [15][1]. A nice summary of Gosper's ideas is given in [46]. Then Vuillemin [47] developed Gosper's work further and gave a representation of $\mathbb{R}_c$ by infinite product of homographies. Vuillemin's work is then further developed by [33, 31] and more recently by [38, 37] where a computable real number is represented by, what they called, an exact floating point. It is the representation given in [38] that we will describe here.

This chapter is organised as follows. First, we summarize Gosper's ideas as described in [46]. We then describe Vuillemin's representation of $\mathbb{R}_c$. Finally we describe the representation of $\mathbb{R}_c$ by `lft`.

4.2 Gosper's Continued Fraction Arithmetic

Gosper observed that performing arithmetic with continued fraction can be manageable if we treat the continued fraction as a formal symbol and analyse the worst case that can happen during the calculation. His ideas can be summarized as follows:

- the worst form that can happen in a unary operation is a linear fractional form (a (2 by 2) matrix).
- the worst form that can happen in a binary operation is a bilinear fractional form (a pair of (2 by 2) matrices or a tensor).

[1] see also [17]

4.2.1 linear fractional form (lff)

Gosper's first idea can be explained by the following example. We want to multiply $14/11 = [1;3,1,2]$ by 3 where we denote the decimal point by a semicolon. Hence

$$\begin{aligned} 3\ [1;3,1,2] &= 3\left(1+\frac{1}{[3,1,2]}\right) \\ &= \frac{3\ [3,1,2]+3}{[3,1,2]} \end{aligned}$$

Observe that the result is an expression of an `lff`

$$\frac{ax+b}{cx+d}$$

by letting $a=3, b=3, c=1, d=0$ and $x=[3,1,2]$. Furthermore it is the worst form that can happen in simple unary operations such as above.

4.2.2 inputing a digit into an lff: $x \mapsto q + \frac{1}{x}$

Gosper's second idea can also be explained by the above example. Observe that $[1;3,1,2]$ transformed into $1+\frac{1}{[3,1,2]}$ or in symbolic form

$$\boxed{x \mapsto q + \tfrac{1}{x}}$$

which is a general phenomenon and can be regarded as inputting a continued fraction digit into an `lff`. Applying this transformation to an `lff`

$$\frac{ax+b}{cx+d} \mapsto \frac{aqx+bx+a}{cqx+dx+c}.$$

We can represent an `lff` as a matrix, then the transformation corresponding to inputting a digit corresponds to

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix} \mapsto \begin{pmatrix} aq+b & a \\ cq+d & c \end{pmatrix} = \begin{pmatrix} a & b \\ c & d \end{pmatrix}\begin{pmatrix} q & 1 \\ 1 & 0 \end{pmatrix}$$

Hence inputting a digit to an `lff` is to perform the above transformation and corresponds to multiplying by $\begin{pmatrix} q & 1 \\ 1 & 0 \end{pmatrix}$ from the right.

4.2.3 outputing a digit from an lff: $x \mapsto x - \frac{1}{q}$

Gosper's third idea is to realize that you can output continued fraction coefficients without having complete knowledge of x. In this case the corresponding transformation is

$$\boxed{x \mapsto x - \tfrac{1}{q}}$$

and the corresponding transformation is

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix} \mapsto \begin{pmatrix} c & d \\ a-cq & b-dq \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & -q \end{pmatrix} \begin{pmatrix} a & b \\ c & d \end{pmatrix}$$

Hence outputing a digit from an `lff` corresponds to multiplying the `lff` by $\begin{pmatrix} 0 & 1 \\ 1 & -q \end{pmatrix}$ from the left.

4.2.4 binary operations on `lff`

Gosper also observed that the worst form that can happen in binary operations is of a *bilinear fractional form* (`bff`). For example, if we perform $x + y$ then we will have the `bff`

$$\frac{axy + bx + cy + d}{exy + fx + gy + h}.$$

where x and y are given as continued fractions. This can be represented by an ordered pair of matrices or a tensor

$$\begin{pmatrix} a & b & e & f \\ c & d & g & h \end{pmatrix}$$

We can input a digit to a `bff` either from x or y. Inputing from x is to perform $x \mapsto q + \frac{1}{x}$ and corresponds to multiplying by $\begin{pmatrix} q & 1 \\ 1 & 0 \end{pmatrix}$ from the left

$$\begin{pmatrix} a & b & e & f \\ c & d & g & h \end{pmatrix} \mapsto \begin{pmatrix} q & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} a & b & e & f \\ c & d & g & h \end{pmatrix}$$

and inputing from y corresponds to multiplying by $\begin{pmatrix} q & 1 \\ 1 & 0 \end{pmatrix}$ from the right.

$$\begin{pmatrix} a & b & e & f \\ c & d & g & h \end{pmatrix} \mapsto \begin{pmatrix} a & b & e & f \\ c & d & g & h \end{pmatrix} \begin{pmatrix} q & 1 \\ 1 & 0 \end{pmatrix}$$

Outputing a digit from a `bff` is almost same as outputing a digit from an `lff`. Observing these Gosper gave algorithms for adding, subtracting, multiplying and dividing continued fractions.

$$\begin{aligned} x + y &= \begin{pmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 \end{pmatrix} (x, y) \\ x - y &= \begin{pmatrix} 0 & 1 & 0 & 0 \\ -1 & 0 & 0 & 1 \end{pmatrix} (x, y) \\ x \times y &= \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} (x, y) \\ x \div y &= \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix} (x, y) \end{aligned}$$

4.3 Vuillemin's Exact Real Arithmetic

Based on Gosper's work, Vuillemin gave a representation of $\mathbb{R}_c$ by redundant continued fractions and algorithms for algebraic and transcendental operations. The key idea of his work is based on the fact that redundant continued fractions solve the two main problems of representing $\mathbb{R}_c$ in terms of interval arithmetic: non-incrementality and size swell. Vuillemin also overcame the fundamental block of the uncomputability of the integer part of a continued fraction by adopting an approximate ordering, similar to `separate` described in Chapter 3, and used it to define a computable rounding operation which he called *Euclidean part* of a computable real number.

4.3.1 a computable rounding

Definition 4.3.1 ($\mathbb{N}$-continued fraction (`ncf`)) *The $\mathbb{N}$-continued fraction of a real number $x = x_0$ is the continued fraction expansion using floor*

$$x_0 = k_0 + \frac{1}{x_1}, x_1 = k_1 + \frac{1}{x_2}, \ldots, x_n = k_n + \frac{1}{x_{n+1}}$$

where $k_0 = \lfloor x \rfloor, k_1 = \lfloor x_1 \rfloor, , \ldots, k_n = \lfloor x_n \rfloor$ and the `ncf` *of x is*

$$x \;=\; [k_0 k_1 \cdots k_n] x_{n+1}$$

If we use round instead of floor then we get

Definition 4.3.2 ($\mathbb{Z}$-continued fraction (`zcf`)) *The $\mathbb{Z}$-continued fraction of a real number x is the continued fraction expansion using round*

$$x = k_0 + \frac{1}{x_1}, x_1 = k_1 + \frac{1}{x_2}, \ldots, x_n = k_n + \frac{1}{x_n + 1}$$

where $k_0 = \lfloor x_0 \rceil, k_1 = \lfloor x_1 \rceil, , \ldots, k_n = \lfloor x_n \rceil$ and the `zcf` *of x is*

$$x \;=\; [k_0 k_1 \cdots k_n] x(n+1)$$

Hurwitz [?] showed the following theorem about `zcf`.

Theorem 4.3.3 (Hurwitz) *Let $[k_0 k_1 \cdots k_n \cdots]$ be the* `zcf` *of a computable real number x. Then*

1. *all terms except the first one are integers either ≥ 2 or ≤ -2, i.e.,*

$$0 < n \;\Rightarrow\; |k_n| \geq 2. \tag{4.1}$$

   Furthermore, if $|k_n| = 2$ for some n, then k_{n+1} has the same sign as k_n

2. *if the* `zcf` *is finite, then its last term is not -2*

Table 4.1: ncfs of some famous constants

	ncf
ϕ	$[1\ 1\ 1\ 1\cdots] = [,1]$
$\sqrt{2}$	$[1\ 2\ 2\ 2\cdots] = [1,2]$
$\sqrt{3}$	$[1\ 1\ 2\ 1\ 2\ 1\ 2\cdots] = [1,1\ 2]$
e	$[2\ 1\ 2\ 1\ 1\ 4\ 1\ 1\ 6\ 1\cdots] = [2,1\ (2n+2)\ 1]$
π	$[3\ 7\ 15\ 1\ 292\ 1\ 1\ 1\ 2\ 1\cdots]$

Table 4.2: zcfs of some famous constants

	zcf
ϕ	$[2\ (-3)\ 3\ (-3)\ 3\cdots] = [2,(-3)\ 3]$
$\sqrt{2}$	[1,2]
$\sqrt{3}$	$[2\ (-4)\ 4\ (-4)\ 4\cdots] = [2,(-4)\ 4]$
e	$[3\ (-4)\ 2\ 5\ (-2)\ (-7)\ 2\ 9\ (-2)\cdots] = [3\ (-4),2\ (4n+5)\ (-2)\ (-4n-7)]$
π	$[3\ 7\ 16\ (-294)\ 3\ (-3)\cdots]$

Tables ?? show some examples of ncf and zcf. Note that we used comma to mark the beginning of a periodic sequence such as the ncf $[1,1\ 2] = [1\ 1\ 2,1\ 2] = [1\ 1,2\ 1] = [1,1\ 2\ 1\ 2]$ for $\sqrt{3}$ above. Also when an expression including the variable n appears to the right of the comma then the expression should be repeated for each successive nonnegative integer values, $0, 1, 2, \cdots$.

But both ncf and zcf are uncomputable due to the undecidability of zero. Instead Vuillemin observed that the following is computable.

Definition 4.3.4 (Euclidean part) *The Euclidean part k of a computable real x is an integer computed in finite time, such that*

$$\triangle(x,k) = \frac{2\,|x-k|}{\sqrt{(1+x^2)(1+k^2)}} < 1.$$

Definition 4.3.5 (𝔼-continued fraction (ecf))

$$\texttt{ecf}(x) \;=\; [k_0 k_1 \cdots k_n k_{n+1} \cdots]$$

The Euclidean part k can be computed by the following algorithm.

Algorithm 4.3.6 *Compute an approximant interval $[i(n), s(n)]$ such that $\triangle(s(n), i(n)) < 1$.*

$$\texttt{Ep}(x) \;:=\; \begin{cases} 0 & \text{if } 0 \in [i(n), s(n)] \\ \lfloor i(n) \rceil & \text{if } 0 \notin [i(n), s(n)] \text{ and } ||\lfloor i(n) \rceil|| \le ||\lfloor s(n) \rceil|| \\ \lfloor s(n) \rceil|| & \text{if } 0 \notin [i(n), s(n)] \text{ and } ||\lfloor i(n) \rceil|| > ||\lfloor s(n) \rceil|| \end{cases}$$

Note that both the Eucildean part and ecf are not unique. We denote the $[k_n k_{n+1} \cdots]$ part by x_n. Also note that any *infinite* zcf is also an ecf. Hence the ecf of the Euler number e is the same its zcf.

And to get **zcf** from a **ecf** Vuillemin gave a normalization rule to convert **ecf** into **zcf** and the resulting continued fraction is called $\mathbb{Z}_\infty$-continued fraction (**necf** for normalized Euclidean continued fraction).

Definition 4.3.7 (necf) *Let $x = [k_0 k_1 \cdots k_{n-1}]x(n)$ be a partial* **ecf**, *of length $n > 3$. Then the continued fraction $[k'_0 k'_1 \cdots k'_{n-1}] < k'_n k'_{n+1} k'_{n+2} > \pm x(n)$ is an equivalent* **necf** *where $[k'_0 k'_1 \cdots k'_{n-1}]$ is a* **zcf**. *The normalization rule, norm, is defined as below*

$$
\begin{aligned}
&norm(k_t, k_{t+1}, k_{t+2}, x(t+3), n) := \\
&\qquad \langle k_t k_{t+1} k_{t+2} \rangle r(t+3) \qquad \text{if } n \leq 0 \\
&\qquad norm(k_t + k_{t+2}, k_{t+3}, k_{t+4}, x(t+5), n-2) \qquad \text{if } k_{t+1} = 0 \\
&\qquad norm(k_t + 1, -k_{t+2} - 1, -k_{t+3}, -x(t+4), n-1) \qquad \text{if } k_{t+1} = 1 \\
&\qquad norm(k_t - 1, -k_{t+2} + 1, -k_{t+3}, -x(t+4), n-1) \qquad \text{if } k_{t+1} = -1 \\
&k_t + 1 norm(-2, k_{t+2} + 1, k_{t+3}, x(t+4), n-1) \text{if } k_{t+1} = 2 \text{ and } k_{t+1} k_{t+2} < 0 \\
&k_t - 1 norm(2, k_{t+2} - 1, k_{t+3}, x(t+4), n-1) \text{if } k_{t+1} = -2 \text{ and } k_{t+1} k_{t+2} < 0 \\
&k_t norm(k_{t+1}, k_{t+2}, k_{t+3}, x(t+4), n-1) \text{otherwise}
\end{aligned}
$$

For example, $x = [30\ 0\ (-3)\ 5\ 2 \cdots]$ is an **ecf** and applying the normalization rule we get $norm(30, 0, -3, x(4), n) = [27\ 5\ 2 \cdots]$ which is an **necf**. Note that a comparison algorithm between two **necf**s is also given in [47], more or less similar to the **separate** operation of $\mathbb{R}_{ldr}$, which we will not describe.

4.3.2 $\mathbb{R}_c$ as infinite products of homographies

To solve the problem of spatial inefficiency of stream representation of **necf** Vuillemin came up with a idea that one can represent the first $[k_0 \cdots k_n]$ part by a 2×2 matrix

$$h_i = [k_0 \cdots k_i] = \begin{pmatrix} k_0 & 1 \\ 1 & 0 \end{pmatrix} \cdots \begin{pmatrix} k_n & 1 \\ 1 & 0 \end{pmatrix} = \begin{pmatrix} N_n & N_{n-1} \\ D_n & D_{n-1} \end{pmatrix}.$$

Thus a real number x is represented by an infinite product of 2×2 matrices which are often called *homographies*

$$x \quad = \quad \langle h_0 \cdots h_n \rangle x(n)$$

4.3.3 Algebraic Algorithm

First, the additive inverse and the multiplicative inverse are straightforward to define. Let $x = \langle h_0 \cdots h_n \rangle x(n) = \begin{pmatrix} k_0 & 1 \\ 1 & 0 \end{pmatrix} \cdots \begin{pmatrix} k_n & 1 \\ 1 & 0 \end{pmatrix}$.

Definition 4.3.8 (x^{-1})

$$x^{-1} \quad := \quad \begin{cases} \langle h_1 \cdots h_n \rangle x(n) & \text{if } h_0 = 0 \\ \langle h_0 \cdots h_n \rangle x(n) & \text{if } h_0 \neq 0 \end{cases}$$

Definition 4.3.9 ($-x$)

$$-x \;:=\; \langle -h_0 \cdots - h_n \rangle - x(n)$$

Next, to define addition and multiplication Vuillemin gave a generic algorithm which can be used both for the addition and multiplication. Let $f(r)$ be a quotient of two polynomials with integer coefficients with degree k[2]. First we need a notion of f monotonically increasing ($f_I \nearrow$) and decreasing ($f_I \searrow$) in a given rational interval I.

Definition 4.3.10 ($f_I \nearrow, f_I \searrow$)

$$\begin{aligned} f_I \nearrow &:= \forall [i,s] \subset I.\ f[i,s] = [f(i), f(s)] \\ f_I \searrow &:= \forall [i,s] \subset I.\ f[i,s] = [f(s), f(i)] \end{aligned}$$

We say that f_I is monotonic if it is either $f_I \nearrow$ or $f_I \searrow$. Note that if f_I is $\nearrow$ (resp. $\searrow$) then the functions $x \mapsto f(k + \frac{1}{x})$ and $x \mapsto \frac{1}{k+f(x)}$ are $\searrow$ (resp. $\nearrow$). We also define $\perp$ as an undefined number. $\perp$ is not an `ecf`.

Definition 4.3.11 ($\perp$)

$$\perp := \frac{0}{0} = 0 \times \infty = 0^0 = \infty - \infty$$

Definition 4.3.12 (Algebraic Algorithm for an `ecf`) *Let $x = [k_0 k_1 \cdots]$ be an* `ecf` *$\neq \perp$ and assume that $f_{(|k_0|-\frac{1}{2}, \frac{1}{2}-|k_0|)}$ is monotonic. $y = [l_0 l_1 \cdots] = f(x)$ is computed as follows. Assume that at time $t = n + m$, we have produced n terms of y by looking at m terms of x. Assume that we are currently computing $l_n = r_t(k_m)$, and write this as*

$$[l_0 \cdots l_{n-1}]\ r_t\ [k_m k_{m+1} \cdots]$$

Let $i = f(|k_m| - \frac{1}{2})$ and $s = f(\frac{1}{2} - |k_m|)$.

- *if $\triangle(i,s) < 1$, then output $l_n := \mathrm{Ep}(x)$, i.e.,*

  $$[l_0 \cdots l_{n-1} l_n]\ f_{t+1}\ [k_m k_{m+1} \cdots]$$

  and

  $$f_{t+1} \;:=\; x \mapsto \frac{1}{f_t(x) - l_n}$$

- *otherwise, consume k_m, i.e.,*

  $$[l_0 \cdots l_{n-1}]\ f_{t+1}\ [k_{m+1} \cdots]$$

  and

  $$f_{t+1} \;:=\; x \mapsto f_t(k_m + \frac{1}{x})$$

For special cases such as multiplying a real by a rational, this algorithm is very inefficient in comparison to the homographic version below.

[2]Note that if the degree of $f(r)$ is one, then we have a homography

4.3.4 homographic algorithm

As mentioned, a homography is a quotient of polynomials with degree one, i.e., $\frac{Nx+n}{Dx+d}$. Addition and multiplication by a rational number are special cases of homographies as we can see below

$$\begin{aligned}\frac{n}{d}+x &:= \begin{pmatrix} d & n \\ 0 & d \end{pmatrix} x \\ \frac{n}{d}\times x &:= \begin{pmatrix} n & 0 \\ 0 & d \end{pmatrix} x\end{aligned}$$

Definition 4.3.13 (Homographic Algorithm) *Let $x = [x_t x_{t+1} \cdots]$ and $h_t = \frac{Nx_t+n}{Dx_t+d}$. Then $y = h(x)$ is computed as follows.*

$$[y_0 \cdots y_{t-1}][h_t][x_t x_{t+i} \cdots] \;\mapsto\; [y_0 \cdots y_{t-1} y_t][h_{t+1}][x_{t+i} \cdots]$$

where

$$\begin{aligned} h_i' &:= x \mapsto h_i(x_t + \frac{1}{x}) = \begin{pmatrix} N_i & n_i \\ D_i & d_i \end{pmatrix}\begin{pmatrix} x_i & 1 \\ 1 & 0 \end{pmatrix} = \begin{pmatrix} N_i' & n_i' \\ D_i' & d_i' \end{pmatrix} \\ y_i &:= \lfloor h_i'(\infty) \rfloor = \lfloor N_i'/D_i' \rfloor \\ h_{i+1} &:= x \mapsto \frac{1}{h_i'(x) - y_i} = \begin{pmatrix} 0 & 1 \\ 1 & -y_i \end{pmatrix}\begin{pmatrix} N_i' & n_i' \\ D_i' & d_i' \end{pmatrix} = \begin{pmatrix} N_{i+1} & n_{i+1} \\ D_{i+1} & d_{i+1} \end{pmatrix}\end{aligned}$$

As an example let us calculate the `ecf` of $10e$. Note that $\texttt{ecf}(e) = [3\ (-4)\ 2\ 5\ (-2)\ (-7) \cdots]$ and thus $10e = \begin{pmatrix} 10 & 0 \\ 0 & 1 \end{pmatrix}[3\ (-4)\ 2\ 5\ (-2)\ (-7) \cdots]$. First, we have

$$\begin{pmatrix} 10 & 0 \\ 0 & 1 \end{pmatrix}\begin{pmatrix} 3 & 1 \\ 1 & 0 \end{pmatrix} = \begin{pmatrix} 30 & 10 \\ 1 & 0 \end{pmatrix}.$$

Hence

$$\begin{aligned} h_0' &= \begin{pmatrix} 30 & 10 \\ 1 & 0 \end{pmatrix} \\ y_0 &= 30/1 = 30 \\ h_1 &= \begin{pmatrix} 0 & 1 \\ 1 & -30 \end{pmatrix}\begin{pmatrix} 30 & 10 \\ 1 & 0 \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & 10 \end{pmatrix}\end{aligned}$$

Continuing like this we get $10e = [30\ 0\ (-3)\ 5\ 2\ \cdots]$ which, after normalization, is equivalent to $[27\ 5\ 2\ \cdots]$.

4.3.5 quadratic algorithm

The algebraic algorithm can be generalised to polynomials in two variables, $\frac{(Nx+N')y+(nx+n')}{(Dx+D')y+(dx+d')}$.

Definition 4.3.14 (Quadratic Algorithm) *Let $x = [x_0 x_1 \cdots], y = [y_0 y_1 \cdots]$ and $h = \frac{(Nx+N')y+(nx+n')}{(Dx+D')y+(dx+d')}$. Then $z = h(x, y)$ is computed as follows.*

$$[z_0 \cdots z_{2i-1}][h_{2i}]([x_i x_{i+i} \cdots], [y_i y_{i+1} \cdots]) \;\mapsto\; [z_0 \cdots z_{2i-1} z_i z_{2i+1}][h_{2i+2}]([x_{i+1} \cdots], [y_{i+1} \cdots])$$

where

$$\begin{aligned} z_{2i} &:= \lfloor h_{2i}(x_i, \infty) \rceil \\ h_{2i+1}(x,y) &:= (x,y) \mapsto \frac{1}{h_{2i}(x_i + 1/x, y) - z_{2i}} \\ z_{2i+1} &:= \lfloor h_{2i+1}(\infty, y_i) \rceil \\ h_{2i+2}(x,y) &:= (x,y) \mapsto \frac{1}{h_{2i+1}(x, y_i + \frac{1}{y_i}) - z_{2i+1}}. \end{aligned}$$

In practice, z_i is computed by merging x and y into one continued fraction $X = [x_0 y_0 x_1 y_1 \cdots]$ and representing h by a 2×4 matrix $\begin{pmatrix} N_i & N_i' & n_i & n_i' \\ D_i & D_i' & d_i & d_i' \end{pmatrix}$.

$$\begin{aligned} z_{2i} &:= \lfloor \tfrac{N_i X_i + N_i'}{D_i X_i + D_i'} \rceil \\ h_{i+1} &:= \begin{pmatrix} D_i X_i + D_i' & D_i & d_i X_i + d_i' & d_i \\ N_i X_i + N_i' - zi(D_i X_i + D_i') & N_i - ziD_i & n_i X_i + n_i' - zi(d_i X_i + d_i') & n_i - zid_i \end{pmatrix} \end{aligned}$$

As a special case of the quadratic algorithm, the sum and product of two continued fractions is compued as follows

$$[x_0 x_1 x_2 \cdots] + [y_0 y_1 y_2 \cdots] := [x_0 + y_0] \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 \end{pmatrix} [x_1 y_1 x_2 y_2 \cdots]$$

$$\mathrm{x}_0 x_1 x_2 \cdots \times [y_0 y_1 y_2 \cdots] := [x_0 \times y_0] \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & y_0 & x_0 & 0 \end{pmatrix} [x_1 y_1 x_2 y_2 \cdots]$$

4.3.6 transcendental functions

[47] also gave algorithms for several transcendental functions $f(x) = [f_1(x) f_2(x) \cdots]$, where each f_i is a function of x. For example, $e(x)$ is computed by using the formula given by Legendre

$$e(x) = [, 1\ \frac{2n+1}{x} - 1\ 1] = [1, \frac{4n+1}{x}\ 2, -\frac{4n+3}{x}\ (-2)]$$

and logarithm is computed by

$$\log(1+x) = [0, \frac{2n+1}{x}\ \frac{2}{x+1}]$$

For other functions, see [47].

We regret that we didn't have time to implement Vuillemin's model into Axiom and compare the performance with Potts and Edalat's model. Vuillemin's work has since been superseded by many others [33, 31, 38, 37].

4.4 Potts and Edalat's Exact Real Arithemtic: $\mathbb{R}_{lft}$

Potts and Edalat proposed an exact real arithmetic based on linear fractional transformation (`lft`). Their representation of $\mathbb{R}_c$ is based on the special base interval $[0, \infty]$ in the one-point compactification $\mathbb{R}^\star$ of the real line $\mathbb{R}$.

In a representation which they call **exact floating point** (`efp`), a real number is represented by a possibly infinite product of `lfts` where the first matrix is one of the four predetermined sign matrices with integer entries and the rest are one of the three predetermined digit matrices which have positive integer entries. This product corresponds to a shrinking sequence of nested rational intervals which are generated by applying the composition of the `lfts` to the real number. `efp` can be thought of a combination of the signed binary (hence base 2) representation and `lft`. Functions are then represented by so-called expression trees of `lfts` whose nodes have `lfts` with 0, 1 or 2 arguments.

4.4.1 $\mathbb{R}_c$ in $\mathbb{R}_{lft}$

Let us start with some notations and preliminary definitions.

Definition 4.4.1 (vector: $\mathbb{V}$)

$$\mathbb{V} := \left\{ \begin{pmatrix} a \\ b \end{pmatrix} \mid a, b \in \mathbb{Z} \right\}$$

Definition 4.4.2 (matrix: $\mathbb{M}$)

$$\mathbb{M} := \left\{ \begin{pmatrix} a & b \\ c & d \end{pmatrix} \mid a, b, c, d \in \mathbb{Z} \right\}$$

Definition 4.4.3 (tensor: $\mathbb{T}$)

$$\mathbb{T} := \left\{ \begin{pmatrix} a & b & c & d \\ e & f & g & h \end{pmatrix} \mid a, b, c, d, e, f, g, h \in \mathbb{Z} \right\}$$

An `lft` is either a vector, a matrix or a tensor.

Definition 4.4.4 (`lft`)

$$\texttt{lft} := \mathbb{V} \cup \mathbb{M} \cup \mathbb{T}$$

After fixing the base interval[3] to be $[0, \infty]$, define the following.

Definition 4.4.5 (Information of an `lft` L: `info`)

$$\texttt{info}(L) := \begin{cases} [a, b] & \text{if } L = V = \begin{pmatrix} a \\ b \end{pmatrix}, V \in \mathbb{V} \\ M[0, \infty] & \text{if } L = M, M \in \mathbb{M} \\ T([0, \infty], [0, \infty]) & \text{otherwise} \end{cases}$$

Note that $M[0, \infty] = [M0, M\infty]$ if $det(M) > 0$ and $M[0, \infty] = [M\infty, M0]$ if $det(M) < 0$.

[3] Another possible base is $[-1, 1]$

Definition 4.4.6 ($\mathbb{V}^+$)

$$\mathbb{V}^+ := \{V \in \mathbb{V} \mid V[0,\infty] \subseteq [0,\infty]\}$$

Definition 4.4.7 ($\mathbb{M}^+$)

$$\mathbb{M}^+ := \{M \in \mathbb{M} \mid M[0,\infty] \subseteq [0,\infty]\}$$

Definition 4.4.8 ($\mathbb{T}^+$)

$$\mathbb{T}^+ := \{T \in \mathbb{T} \mid T([0,\infty],[0,\infty]) \subseteq [0,\infty]\}$$

In $\mathbb{R}_{lft}$, with digit base 2 and interval base $[0,\infty]$, a real number x is represented as a sequence of one-dimensional `lfts`, $n \mapsto M_0 \ldots M_n[0,\infty]$, where M_0 is a sign matrix and $M_i, i \geq 1$ are digits matrices.

Definition 4.4.9 (sign matrices: S_+, S_∞, S_-, S_0)

$$S_+ := \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \quad S_\infty := \begin{pmatrix} 1 & 1 \\ -1 & 1 \end{pmatrix}$$
$$S_- := \begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix} \quad S_0 := \begin{pmatrix} 1 & -1 \\ 1 & 1 \end{pmatrix}$$

Definition 4.4.10 (digit matrices: D_{-1}, D_0, D_1)

$$D_{-1} := \begin{pmatrix} 1 & 0 \\ 1 & 2 \end{pmatrix} \quad D_0 := \begin{pmatrix} 3 & 1 \\ 1 & 3 \end{pmatrix} \quad D_1 := \begin{pmatrix} 2 & 1 \\ 0 & 1 \end{pmatrix}$$

One can easily find that

$$\texttt{info}(S_+) = [0,\infty] \quad \texttt{info}(S_\infty) = [-1,1]$$
$$\texttt{info}(S_-) = [\infty,0] \quad \texttt{info}(S_0) = [-1,1]$$

$$\texttt{info}(D_{-1}) = [0,1] \quad \texttt{info}(D_0) = [1/3,3] \quad \texttt{info}(D_1) = [1,\infty]$$

In $\mathbb{R}_{lft}$, everything is represented by expression trees whether it is a real number or a real function. A rational number is represented by a vector, unary functions by matrices, and binary operations (and transcendental functions) by tensors. Hence the definition of expression trees include all these possibilities.

Definition 4.4.11 (unsigned expression tree (uexp)) *An unsigned expresssion tree is either a vector V where $V \in V^+$, or a $M(uexp)$, where $M \in M^+$, or a $T(uxep, uexp)$, where $T \in T^+$.*

Definition 4.4.12 (signed expression tree (sexp)) *A signed expression tree is either a vector V where $V \in V$, or a $M(uexp)$, where $M \in M$, or a $T(uxep, uexp)$, where $T \in T$.*

An integer k is represented by a vector expression tree $(k, 1)$.

4.4.2 functions in $\mathbb{R}_{lft}$

The additive and multiplicative inverse operations of a real number which is an expression tree are represented by *matrix applications* to the expression tree. Application of a matrix (or a tensor) to an expression tree is defined as the product of the matrix (or the tensor) and the root node, i.e. an `lft`, of the expression tree.

Definition 4.4.13 (matrix application (mapp)) *Let $M \in M$ be a matrix and E be an expression tree with the* `lft` *L as its root node. Then*

$$mapp(M, E) \quad := \quad (M \times L)E$$

Let E be an expression tree which represents a real number.

Definition 4.4.14 (additive inverse) $-E \;:=\; mapp(\begin{pmatrix} -1 & 0 \\ 0 & 1 \end{pmatrix}, E)$

Definition 4.4.15 (multiplicative inverse) $-E \;:=\; mapp(\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}, E)$ The four arithmetic operations are represented by *tensor applications* to the tvl $\begin{pmatrix} \\ 0 \end{pmatrix}, E)$ The four originally worked out by Gosper.

Definition 4.4.16 (tensor application (tapp)) *Let $T \in T$ be a tensor and E and F be two signed expression trees with L_E and L_F as their root nodes respectively and $L_E, L_F \in V \cup M$. Then*

$$tapp(T, E, F) \quad := \quad ((T \times_2 L_F) \times_1 L_E)(E')$$

where E' is the new expression tree formed by attaching E in front of F.

The products $M \times L$ and $\times_2, \times_1$ above are defined in [38]. The definitions of $+, -, \times, \div$ are then as below.

$$\begin{aligned}
E + F &:= tapp(\begin{pmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}, E, F) \\
E - F &:= tapp(\begin{pmatrix} 0 & 1 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}, E, F) \\
E \times F &:= tapp(\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}, E, F) \\
E \div F &:= tapp(\begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix}, E, F)
\end{aligned}$$

Transcendental functions are represented by expression trees, too. For details see Potts's thesis [38].

4.5 Heckmann's Work on $\mathbb{R}_{lft}$

Heckmann [20, 22, 21, 24, 23] did a series of work on $\mathbb{R}_{lft}$. Here we only give a brief summary of his work as follows.

- (appearance of big integers): in [20], Heckmann observed that the size of the biggest entry in a matrix or a tensor usually increases with the number of digits emitted and absorbed.
- (contractivity): in [22], Heckmann defined a notion of contractivity for `lfts` and use it for convergence analysis.
- (input complexity): in [24], Heckmann analysed the number of argument (input) digits necessary to produce n result (output) digits and gave lower and upper bounds for many functions.
- (derivation of `lft` from Taylor series) : Potts derived expression trees of transcendental functions from continued fraction expansions. In [23], Heckmann derived these expression trees from series expansions.

Chapter 5

A Category for $\mathbb{R}_C$ in Axiom

5.1 Introduction

$\mathbb{R}_c$ has several different models. How do we model these various views of computable reals in Axiom? As discussed in Chapter 2, $\mathbb{R}_c$ is a field in which zero is undecidable [40]. In other words, $\mathbb{R}_c$ is a field without **equality** [12]. In this chapter we build a category, `LazyOrderedField`, for such fields without equality. Axiom[1] has a built-in category called `Field` but this is a category of fields where we can test equality. So every domain in `Field` has equality testing. So we can not define $\mathbb{R}_c$ as a domain of `Field` category but have to build ourselves a new category for $\mathbb{R}_c$. Note that although $\mathbb{R}_c$ doesn't have equality it does have a notion of order. In view of this we will name the category as `LazyOrderedField`. Since we have a notion of order, we are assuming that the ring of computable real numbers are of characteristic zero. Thinking of defining a new category we have to answer to the fundamental question:

> What is a suitable category for $\mathbb{R}_c$, i.e, what operations can reasonably be generic across the various instantiations of $\mathbb{R}_c$?

We answer this question by filtering out those operations of `Field` which are, either directly or indirectly, related to equality.

Equality is important. For example, definition of determinant doesn't need it, but Gaussian elimination does. So removing equality will make the complexity of Gaussian elimination from $O(n^3)$ to $O(n!)$.

This chapter is organized as follows.

- section 5.2: we define a category `LazyOrderedField` for $\mathbb{R}_c$.
- section 5.3: we define a category `LazyReal` as an extended category of `LazyOrderedField`.
- section 5.4: we define `LDR` for $\mathbb{R}_{ldr}$ and `LFT` for $\mathbb{R}_{lft}$ as domains of `LazyReal`.
- section 5.5: we define a category `Pair` for the particular symbolic-numeric pair representation of real algebraic number which will be described in Chapter 6.

[1] We will not desribe Axiom's type structure [1, 25] in detail. Instead we will briefly mention various points when necessary.

- section 5.6: we define `LDRPair` and `LFTPair` as domains of `Pair`.

We will use the following abbreviations for frequently used domains.

Axiom Domain	Abbreviation
`Boolean`	`B`
`Integer`	`I`
`Fraction I`	`Q`
`SingleInteger`	`SI`
`PositiveInteger`	`PI`
`NonNegativeInteger`	`NI`
`SparseUnivariatePolynomial`	`SUP`

5.2 LazyOrderedField: a Category for $\mathbb{R}_c$

What should a category for $\mathbb{R}_c$ look like? In other words, what kind of operations can reasonably be generic across the various instantiations of $\mathbb{R}_c$, or, algebraically speaking, what is a field without equality? As a first step to answer these questions we can look at the operations of the Axiom's category for a field with equality, `Field`. Axiom shows us the following operations of `Field`.

```
          * : (Q,%) -> %                          * : (%,Q) -> %
          * : (%,%) -> %                          * : (I,%) -> %
          * : (PI,%) -> %                        ** : (%,I) -> %
         ** : (%,PI) -> %                         + : (%,%) -> %
          - : (%,%) -> %                          - : % -> %
          / : (%,%) -> %                          = : (%,%) -> B
          1 : () -> %                             0 : () -> %
          ^ : (%,I) -> %                          ^ : (%,PI) -> %
associates? : (%,%) -> B                    coerce : Q -> %
     coerce : % -> %                        coerce : I -> %
     coerce : % -> OutputForm               factor : % -> Factored %
        gcd : List % -> %                       gcd : (%,%) -> %
       hash : % -> SI                           inv : % -> %
      latex : % -> String                       lcm : List % -> %
        lcm : (%,%) -> %                       one? : % -> B
    prime? : % -> B                             quo : (%,%) -> %
     recip : % -> Union(%,"failed")             rem : (%,%) -> %
    sample : () -> %                    sizeLess? : (%,%) -> B
 squareFree : % -> Factored %       squareFreePart : % -> %
     unit? : % -> B                  unitCanonical : % -> %
     zero? : % -> B                           ~= : (%,%) -> B
         * : (NI,%) -> %                       ** : (%,NI) -> %
         ^ : (%,NI) -> %           characteristic : () -> NI
    divide : (%,%) -> Record(quotient: %,remainder: %)
    euclideanSize : % -> NI
    expressIdealMember : (List %,%) -> Union(List %,"failed")
    exquo  : (%,%) -> Union(%,"failed")
    extendedEuclidean : (%,%,%) -> Union(Record(coef1: %,coef2: %),"failed")
    extendedEuclidean : (%,%) -> Record(coef1: %,coef2: %,generator: %)
```

```
gcdPolynomial : (SUP %,SUP %) -> SUP %
multiEuclidean : (List %,%) -> Union(List %,"failed")
principalIdeal : List % -> Record(coef: List %,generator: %)
subtractIfCan : (%,%) -> Union(%,"failed")
unitNormal : % -> Record(unit: %,canonical: %,associate: %)
```

These operations are the Axiom's answer for the same question for a field with equality. Hence we can simply identify generic operations for a field without equality by removing those operations of field that are related with equality. For example, following operations of `Field` directly involve equality testing.

```
one? : % -> B                    zero? : % -> B3
unit? : % -> B                       = : (%,%) -> B3
~= : (%,%) -> B
```

Also many other operations, such as `gcd` and `recip`, use or are related to equality.

```
associates? : (%,%) -> B              factor : % -> Factored %
gcd : List % -> %                        gcd : (%,%) -> %
hash : % -> SingleInteger                lcm : List % -> %
lcm : (%,%) -> %                       recip : % -> Union(%,"failed")
squareFree : % -> Factored %      unitCanonical : % -> %
expressIdealMember : (List %,%) -> Union(List %,"failed")
extendedEuclidean : (%,%,%) -> Union(Record(coef1: %,coef2: %),"failed")
extendedEuclidean : (%,%) -> Record(coef1: %,coef2: %,generator: %)
gcdPolynomial : (SUP %,SUP %) -> SUP %
multiEuclidean : (List %,%) -> Union(List %,"failed")
principalIdeal : List % -> Record(coef: List %,generator: %)
unitNormal : % -> Record(unit: %,canonical: %,associate: %)
```

Note that most of the above operations are special cases of `gcd` operation which is originated from the `GcdDomain`. Hence by removing these operations from `Field` we are left with operations which we think that are good candidates for the operations (of an instantiation) of `LazyOrderedField`. So, if we want, we can define `LazyOrderedField` as below.

```
LazyOrderedField : Category == with {
    * : (PI,%) -> %;                    ** : (%,I) -> %;
    ^ : (%,PI) -> %;                     * : (NI,%) -> %;
   ** : (%,NI) -> %;                     ^ : (%,NI) -> %;
    * : (Q,%) -> %;                      * : (%,Q)->%;
    * : (%,%) -> %;                      * : (Z,%) -> %;
   ** : (%,Z) -> %;                      + : (%,%) -> %;
    - : (%,%) -> %;                      - : % -> %;
    / : (%,%) -> %;                      1 : () -> %;
    0 : () -> %;                         ^ : (%,Z) -> %;
  coerce : Q -> %;                    coerce : % -> %;
  coerce : Z -> %;                    coerce : % -> OutputForm;
     inv : % -> %;                     latex : % -> String;
  prime? : % -> B;                       quo : (%,%) -> %;
     rem : (%,%) -> %;                sample : () -> %;
sizeLess? : (%,%) -> B;          characteristic: ()->NI;
  divide : (%,%) -> Record(quotient:%,remainder:%);
```

```
euclideanSize : % -> NI;
exquo  : (%,%) -> Union(exq:%,failed:'failed');
subtractIfCan : (%,%) -> Union(sic:%,failed:'failed');
}
```

This is fine. But it doesn't have a notion of ordering. To include the notion of ordering and to follow Axiom's spirit more faithfully we decided to define all the basic categories from the scratch. To learn how to do this, we trace down Axiom's category hierarchy and we see that the `Field` category is a child of the root category `BasicType`, the basic category for describing a collection of elements with equality. All the algebraic domains of Axiom are children of `BasicType`. `BasicType` exports only two operations:

```
= : (%,%) -> B                    ~= : (%,%) -> B
```

In fact, Axiom defines `SetCategory`, the category of all algebriac domains in Axiom, as a child of `BasicType`. Note that `SetCategory` is also a child of the category `CoercibleTo(S:Type)` which provides the fundamental coercion operation `coerce`.

5.2.1 lazy set

So, what we need is a category, similar to `BasicType`, which can describe a collection of elements without equality. But, although $\mathbb{R}_c$ cannot have exact equality, it does have a weaker notion of (in)equality (lazy order henceforth) as we have seen in Chapter 3. Remembering this we can define a category which we call `LazyBasicType`.

```
LazyBasicType : Category == with {
          separate : (%,%) -> I;
}
```

The above defintion says that `LazyBasicType` is a category and exports one operation called `separate` which we described in Chapter 2 and 3. Note that the `separate` operation and operations based on it may be only partially defined since we can't separate 0 from 0. It gives a notion of ordering for $\mathbb{R}_c$ (hence for any domain of $\mathbb{R}_c$ such as $\mathbb{R}_{ldr}$ and $\mathbb{R}_{lft}$). We will define a category called `LazyOrderedSet` below as a children of `LazyBasicType`.

Imitating Axiom's categorical structure, we define `LazySetCategory`, a category similar to `SetCategory`, which will act as the category of all domains with lazy ordering.

```
LazySetCategory : Category == Join(LazyBasicType,
                                   CoercibleTo OutputForm) with {
          latex : % -> String;
}
```

The `Join` operation is a categorical union and combines the operations from `LazyBasicType` and `CoercibleTo OutputForm` into a single category as we can see below.

```
  coerce : % -> OutputForm          latex : % -> String
separate : (%,%) -> I
```

5.2.2 lazy set with one operation

We also define LazySemiGroup, LazyMonoid, and LazyGroup, similar to Axiom's categories SemiGroup, Monoid, and Group respectively. In doing so, we define two parameterised packages: LazyRepeatedSquaring(S) and LazyRepeatedDoubling(S) where S is a domain of LazySetCategory[2]. LazyRepeatedSquaring implements exponentiation by repeated squaring and LazyRepeatedDoubling implements multiplication by repeated doubling.

```
LazyRepeatedSquaring(S:LazySetCategory with {*:(%,%)->%}): with {
     expt          : (S,PI) -> S;
} == add {
          import from I;
          expt(x:S,n:PI):S == {
                  one? n => x;
                  odd?(n::I) => x * expt(x*x,shift(n::I,-1)::PI);
                  expt(x*x,shift(n::I,-1)::PI)
          }
}

LazyRepeatedDoubling(S:LazySetCategory with {+:(%,%)->%}): with {
     double        : (PI,S) -> S;
} == add {
          import from I;
          double(n:PI,x:S):S == {
                  one?(n) => x;
                  odd?(n::I) => x + double(shift(n::I,-1)::PI,(x+x));
                  double(shift(n::I,-1)::PI,(x+x))
          }
}
```

Now we can define LazySemiGroup, LazyMonoid and LazyGroup which all uses the repeated squaring operation expt.

```
LazySemiGroup : Category == LazySetCategory with {
     *             : (%,%) -> %;
     **            : (%,PI) -> %;
     ^             : (%,PI) -> %;

     import from LazyRepeatedSquaring(%);
     default {
          (x:%)**(n:PI):% == expt(x,n);
          (x:%)^(n:PI):% == x**n;
     }
}

LazyMonoid : Category == LazySemiGroup with {
     1             : %;
     sample        : %;
     **            : (%,NI) -> %;
```

[2]These are exact copy of Axiom's RepeatedSquaring(S) and RepeatedDoubling(S) but these require S to be a domain of SetCategory which is why we have to define their lazy counterparts

```
        ^           : (%,NI) -> %;

        import from LazyRepeatedSquaring(%);
        default {
            sample:% == 1;
            (x:%)**(n:NI):% == {
                    zero?(n) => 1;
                    expt(x, n pretend PI)
            }
            (x:%)^(n:NI):% == x**n;
        }
}

LazyGroup : Category == LazyMonoid with {
   inv          : % -> %;
     /          : (%,%) -> %;
    **          : (%,I) -> %;
     ^          : (%,I) -> %;

     import from LazyRepeatedSquaring(%);
     default {
          (x:%)/(y:%):% == x * inv(y);
          (x:%)^(n:I):% == x**n;
          (x:%)**(n:I):% == {
                  zero?(n) => 1;
                  n<0 => expt(inv x,(-n) pretend PI);
                  expt(x,n pretend PI)
          }
     }
}
```

The `default` operation enables us to use the operations defined inside its scope in all domains belonging to the `LazySemiGroup` category. Axiom treats commutative algebraic structures separately, so we follow this and define few more categories.

```
LazyAbelianSemiGroup : Category == LazySemiGroup with {
     +          : (%,%) -> %;
}

LazyAbelianMonoid : Category == LazyAbelianSemiGroup with {
     0          : %;
}

LazyCancellationAbelianMonoid : Category == LazyAbelianMonoid with {
     subtractIfCan: (%,%) -> Union(diff:%,failed:'failed');
}

LazyAbelianGroup : Category == LazyCancellationAbelianMonoid with {
     -          : % -> %;
     -          : (%,%) -> %;
     *          : (I,%) -> %;
```

```
        import from LazyRepeatedDoubling(%);
        default {
            (x:%)-(y:%):% == x + (-y);
            subtractIfCan(x:%,y:%):Union(diff:%,failed:'failed') ==
                    (x-y) pretend Union(diff:%,failed:'failed');
            (n:I)*(x:%):% == {
                    zero?(n) => 0;
                    n>0 => double(n pretend PI,x);
                    double((-n) pretend PI,-x)
            }
        }
}
```

Note that `LazyAbelianGroup` imports the `double:(PI,S)->S` from `LazyRepeatedDoubling(S)` where `S` must be a domain of `LazySetCategory` with an addition.

5.2.3 lazy set with two operations

The next algebraic structure in line is a ring, a structure with two operations. Again, we shamelessly follow Axiom and define `LazyRing` and related categories as below.

```
LazyRng : Category == Join(LazyAbelianGroup,LazySemiGroup);

LazyLeftModule(R:LazyRng) : Category == LazyAbelianGroup with {
    *           : (R,%) -> %;
}

LazyRightModule(R:LazyRng) : Category == LazyAbelianGroup with {
    *           : (%,R) -> %;
}

LazyRing : Category == Join(LazyRng,
                            LazyMonoid,
                            LazyLeftModule(%)) with {
    coerce              : I -> %;
    characteristic      : NI;
    default {
        characteristic:NI == 0;
        coerce(n:I):% == n*1$%;
    }
}

LazyBiModule(R:LazyRing,S:LazyRing) : Category ==
                    Join(LazyLeftModule(R),
                         LazyRIghtModule(S));

LazyCharacteristicZero : Category == LazyRing;

LazyCharacteristicNonZero : Category == LazyRing with {
    charthRoot   : % -> Union(root:%,failed:'failed');
```

```
}

LazyCommutativeRing : Category == Join(LazyRing,
                                       LazyBiModule(%,%)) with {
     commutative((*:(%,%)->%) pretend Type)
}

LazyModule(R:LazyCommutativeRing) : Category == LazyBiModule(R,R) with {
     default (x:%)*(r:R):% == r*x;
}

LazyAlgebra(R:LazyCommutativeRing) : Category ==
                        Join(LazyRing,
                             LazyModule R) with {
     coerce       : R -> %;
     default coerce(x:R):% == x*1$%;
}
```

The next categories we define are `IntegralDomain` and `DivisionRing`. But we do not define the lazy counterparts of Axiom's `EuclideanDomain` and `UniqueFactorizationDomain` since these require the `gcd` operation.

```
LazyIntegralDomain : Category ==  Join(LazyCommutativeRing,
                                       LazyAlgebra(%)) with {
     exquo        : (%,%) -> Union(eqo:%,failed:'failed');
}

LazyDivisionRing : Category == LazyCommutativeRing with {
     **           : (%,I) -> %;
     ^       : Category == LazyCommutativeRing with {
     inv          : (%,I) -> %;
     coerce       : (%,I) -> %;
     *            : % -> %;
     *            : Q -> %;        <--
                  : (Q,%) -> %;    <--
}                 : (%,Q) -> %;    <--
```

Axiom's `DivisionRing` is defined slightly differently. It is defined as an extended category of both `EntireRing` and `Algebra Q` (in Axiom's language, `Join(EntireRing, Algebra Q)`) to inherit operations involving rational numbers[3]. But we can not join `Algebra Q` since it will inherit not only `Q`-related operations but also others which involves equality. So we defined `LazyDivisionRing` as an extension of `LazyCommutativeRing` alone. But to ensure it exports some `Q`-related operations (we want only three which are marked with `<--` above) we exported them explicitly. Note also that `LazyDivisionRing` imports the exponentiation operation `expt:(S,PI)->S` from `LazyRepeatedSquaring`. `LazyDivisionRing` now exports quite a few operations as we can see below.

```
* : (%,%) -> %                    * : (I,%) -> %
* : (Q,%) -> %                    * : (%,Q) -> %
```

[3] `EntireRing` can be thought of as a commutative ring but with no zero divisors. We don't have to define a `LazyEntireRing`. It is unnecessary.

```
    ** : (%,PI) -> %                    ** : (%,I) -> %
     + : (%,%) -> %                      - : % -> %
     - : (%,%) -> %                      0 : () -> %
     1 : () -> %                         ^ : (%,PI) -> %
     ^ : (%,I) -> %                coerce : % -> OutputForm
coerce : I -> %                    coerce : Q -> %
   inv : % -> %                     latex : % -> String
sample : () -> %                  separate : (%,%) -> I
    ** : (%,NI) -> %                    ^ : (%,NI) -> %
characteristic : () -> NI
subtractIfCan : (%,%) -> Union(diff: %,failed: Enumeration failed)
```

5.2.4 lazy ordering

Next we define a category which exports ordering relations using the **separate** operation from **LazyBasicType**.

```
LazyOrderedSet : Category == LazySetCategory with {
     < : (%,%) -> B;
     > : (%,%) -> B;
    >= : (%,%) -> B;
    <= : (%,%) -> B;
   max : (%,%) -> %;
   min : (%,%) -> %;
}
```

The **LazyOrderedSet** has following operations

```
     < : (%,%) -> B                     <= : (%,%) -> B
     > : (%,%) -> B                     >= : (%,%) -> B
coerce : % -> OutputForm             latex : % -> String
   max : (%,%) -> %                    min : (%,%) -> %
separate : (%,%) -> I
```

We also define a group of categories based on **LazyOrderedSet** aiming to define the **abs** operation.

```
LazyOrderedMonoid : Category == Join(LazyOrderedSet, LazyMonoid);

LazyOrderedAbelianSemiGroup : Category ==
                        Join(LazyOrderedSet,
                             LazyAbelianMonoid);

LazyOrderedAbelianMonoid : Category ==
                        Join(LazyOrderedAbelianSemiGroup,
                             LazyAbelianMonoid);

LazyOrderedCancellationAbelianMonoid : Category ==
                        Join(LazyOrderedAbelianMonoid,
                             LazyCancellationAbelianMonoid);

LazyOrderedAbelianGroup : Category ==
                        Join(LazyOrderedCancellationAbelianMonoid,
```

```
                                LazyAbelianGroup);

LazyOrderedRing : Category == Join(LazyOrderedAbelianGroup,
                                   LazyRing,
                                   LazyMonoid) with {
   abs : %-> %;
}
```

5.2.5 lazy ordered field

At last, we have `LazyOrderedField`, our category for fields without equality.

```
LazyOrderedField : Category == Join(LazyOrderedRing,
                                    LazyDivisionRing,
                                    LazyCharacteristicZero) with {
     import from Record(quotient:%,remainder:%);
     default {
          exquo(x:%,y:%):Union(eqo:%,failed:'failed') ==
                  (x/y) pretend Union(eqo:%,failed:'failed');
          (x:%)/(y:%):% == x*inv(y);
          divide(x:%,y:%):Record(quotient:%,remainder:%) == [x/y,0];
          (x:%) quo (y:%):% == (divide(x,y)).quotient;
          (x:%) rem (y:%):% == 0$%;
     }
}
```

The operations of `LazyOrderedField` are shown below.

```
     * : (%,%) -> %                      * : (I,%) -> %
     * : (Q,%) -> %                      * : (%,Q) -> %
    ** : (%,PI) -> %                    ** : (%,I) -> %
     + : (%,%) -> %                      - : % -> %
     - : (%,%) -> %                      / : (%,%) -> %
     0 : () -> %                         1 : () -> %
     < : (%,%) -> B                     <= : (%,%) -> B
     > : (%,%) -> B                     >= : (%,%) -> B
     ^ : (%,PI) -> %                     ^ : (%,I) -> %
   abs : % -> %                      coerce : % -> OutputForm
coerce : I -> %                      coerce : Q -> %
   inv : % -> %                       latex : % -> String
   max : (%,%) -> %                     min : (%,%) -> %
   quo : (%,%) -> %                     rem : (%,%) -> %
sample : () -> %                    separate : (%,%) -> I
    ** : (%,NI) -> %                        ^ : (%,NI) -> %
divide : (%,%) -> Record(quotient: %,remainder: %)
 exquo : (%,%) -> Union(eqo: %,failed: Enumeration failed)
 subtractIfCan : (%,%) -> Union(diff: %,failed: Enumeration failed)
characteristic : () -> NI
```

These are the operations that we think they are reasonably generic across various instantiations of `LazyOrderedField`.

5.3 LazyReal as an Extension of LazyOrderedField

$\mathbb{R}_c$ is a model of the category `LazyOrderedField`. Based on this, we can define an extended category which will contain various models of $\mathbb{R}_c$, such as $\mathbb{R}_{ldr}$ and $\mathbb{R}_{lft}$, as its domains. We call this category `LazyReal`. We define `LazyReal` as an extension of `LazyOrderedField`.

```
LazyReal : Category == LazyOrderedField with {

    msd   : % -> Z;
}
```

In the above definiton of the category `LazyReal` we simply added an operation, `msd`, to `LazyOrderedField` and declared it as a category.

5.4 LDR ($\mathbb{R}_{ldr}$) as a domain of LazyReal

5.4.1 LDR as a domain of LazyReal

As described in Chapter 3, $\mathbb{R}_{ldr}$ is a model of $\mathbb{R}_c$, or more precisely, of $\mathbb{R}_c^{num}$. Hence we can now define $\mathbb{R}_{ldr}$ as a domain `LDR` of `LazyReal`. We show the operations of `LDR`.

```
      * : (%,%) -> %                    * : (I,%) -> %
      * : (Q,%) -> %                    * : (%,Q) -> %
      * : (%,I) -> %                   ** : (%,PI) -> %
     ** : (%,I) -> %                   ** : (I,I) -> %
      + : (%,%) -> %                    + : (I,I) -> %
      + : (I,%) -> %                    + : (%,I) -> %
      + : (Q,%) -> %                    + : (%,Q) -> %
      - : % -> %                        - : (%,%) -> %
      - : I -> %                        - : Q -> %
      / : (%,%) -> %                    / : (%,I) -> %
      / : (%,Q) -> %                    / : (I,%) -> %
      / : (Q,%) -> %                    0 : () -> %
      1 : () -> %                       < : (%,%) -> B
     <= : (%,%) -> B                    > : (%,%) -> B
     >= : (%,%) -> B                    ^ : (%,PI) -> %
      ^ : (%,I) -> %                  abs : % -> %
coerce : % -> OutputForm            coerce : I -> %
coerce : Q -> %                     coerce : % -> %
   inv : % -> %                        inv : I -> %
   inv : Q -> %                      latex : % -> String
   max : (%,%) -> %                   memo : % -> %
   min : (%,%) -> %                    msd : I -> I
   msd : Q -> I                        msd : % -> I
nthRoot : (%,NI) -> %               nthint : (%,I) -> I
   quo : (%,%) -> %                    rem : (%,%) -> %
sample : () -> %                   separate : (%,%) -> I
  sqrt : % -> %                       view : (%,I) -> String
    ** : (%,NI) -> %                   ** : (Q,I) -> %
     + : (Q,I) -> %                    + : (Q,Q) -> %
```

```
      + : (I,Q) -> %                          ^ : (%,NI) -> %
   divide : (%,%) -> Record(quotient: %,remainder: %)
    exquo : (%,%) -> Union(eqo: %,failed: Enumeration failed)
   nthrat : (%,I) -> Q
   subtractIfCan : (%,%) -> Union(diff: %,failed: Enumeration failed)
  characteristic : () -> NI
```

As we can see above LDR exports quite a few other functions additional to the functions exported by LazyReal. As mentioned in section Chapter 3, LDR has the memo function which makes the evaluation **incremental**.

5.5 Pair: a Category for $\mathbb{R}_{pair}$

$\mathbb{R}_{pair}$[4], a model for $\mathbb{R}_a$, is a field. So we can define a category for $\mathbb{R}_{pair}$ as an extension of the category Field. But the Field category exports many operations, for example gcd related operations, which are unnessary for us. So we define Pair[5] as a Join of categories which, we believe, will serve as generic operations for any *pair*-like domains.

```
Pair : Category == Join(CharacteristicZero,
                        OrderedRing,
                        DivisionRing,
                        RadicalCategory) with {
}
```

Pair category exports following operations.

```
      * : (%,%) -> %                          * : (PI,%) -> %
      * : (I,%) -> %                          * : (Q,%) -> %
      * : (%,Q) -> %                         ** : (%,PI) -> %
     ** : (%,I) -> %                         ** : (%,Q) -> %
      + : (%,%) -> %                          - : % -> %
      - : (%,%) -> %                          0 : () -> %
      1 : () -> %                             < : (%,%) -> B
     <= : (%,%) -> B                          = : (%,%) -> B
      > : (%,%) -> B                         >= : (%,%) -> B
      ^ : (%,PI) -> %                         ^ : (%,I) -> %
    abs : % -> %                           coerce : % -> OutputForm
 coerce : I -> %                           coerce : Q -> %
   hash : % -> SI                             inv : % -> %
  latex : % -> String                         max : (%,%) -> %
    min : (%,%) -> %                    negative? : % -> B
nthRoot : (%,I) -> %                         one? : % -> B
positive? : % -> B                        sample : () -> %
   sign : % -> I                             sqrt : % -> %
  zero? : % -> B                              ~= : (%,%) -> B
      * : (NI,%) -> %                         ** : (%,NI) -> %
      ^ : (%,NI) -> %              characteristic : () -> NI
  recip : % -> Union(value1: %,failed: Enumeration failed)
 subtractIfCan : (%,%) -> Union(value1: %,failed: Enumeration failed)
```

[4]We describe $\mathbb{R}_{pair}$ in detail in the next chapter.

[5]Note that Pair is not necessarily exact. It is the domains of Pair which we define as exact models

5.6 LDRPair as a domain of Pair

5.6.1 LDRPair as a domain of Pair

We define LDRPair as domain of Pair. LDRPair represents $\mathbb{R}_a$ by a pair of symbolic and numeric representation. For the numerical part of LDRPair we use LDR and for the symboilc part we use Axiom's polynomial constructor SparseUnivariatePolynomial(Integer). The operations of LDRPair is shown below.

```
          * : (%,%) -> %                          * : (PI,%) -> %
          * : (I,%) -> %                          * : (Q,%) -> %
          * : (%,Q) -> %                         ** : (%,PI) -> %
         ** : (%,I) -> %                         ** : (%,Q) -> %
          + : (%,%) -> %                          - : % -> %
          - : (%,%) -> %                          / : (%,%) -> %
          0 : () -> %                             1 : () -> %
          < : (%,%) -> B                         <= : (%,%) -> B
          = : (%,%) -> B                          > : (%,%) -> B
         >= : (%,%) -> B                          ^ : (%,PI) -> %
          ^ : (%,I) -> %                        abs : % -> %
    coerce : % -> OutputForm                 coerce : I -> %
    coerce : Q -> %                           hash : % -> SingleInteger
       inv : % -> %                          latex : % -> String
       max : (%,%) -> %                        min : (%,%) -> %
negative? : % -> B                              np : % -> LazyDyadicReal
  nthRoot : (%,NI) -> %                       one? : % -> B
positive? : % -> B                          refine : % -> %
    sample : () -> %                          sign : % -> I
      sqrt : % -> %                          zero? : % -> B
        ~= : (%,%) -> B                          * : (NI,%) -> %
        ** : (%,NI) -> %                         ^ : (%,NI) -> %
    characteristic : () -> NI
    mahler : SUP I -> LazyDyadicReal
     recip : % -> Union(value1: %,failed: Enumeration failed)
        sp : % -> SUP I
     sturm : SUP I -> List SUP I
    subtractIfCan : (%,%) -> Union(value1: %,failed: Enumeration failed)
    variation : List Q -> I
```

As we can see LDRPair has several key operations: **sqrt, nthRoot, sturm**, etc. The **refine** operation plays the key role in reducing the size of polynomial by square-free decomposition.

Chapter 6

An Exact Algebraic Arithmetic

6.1 Introduction

We can exactly calculate, not only real algebraic numbers, $\mathbb{R}_a$, but also some transcendental numbers, $\mathbb{R}-\mathbb{R}_a$[1], up to any precision as we want. This means that we can generate, for example, decimal digits of a real number to as much precision as we want and we are guaranteed that this representation is accurate to given precision[2].

But the current exact models suffer from the fundamental problem of undecidability of zero. These motivated us to ask whether we can solve the equality problem (hence zero problem) for the smaller set of $\mathbb{R}_a$. A real algebraic number, say the positive[3] square root of 2 ($\sqrt{2}$), has two kinds of information associated with it : a numerical value of $1.4142\ldots$ when expanded into decimal and a symbolic expression x^2-2 since it is a root of this polynomial. Hence we solve the equality problem for $\mathbb{R}_a$ by representing an $\alpha \in \mathbb{R}_a$ as a pair of its numerical value and a squarefree polynomial (or possibly a multiple of it) of which one of its roots is the numerical value.

For the numerical part we can use any exact real representation but in this chapter we use $\mathbb{R}_{ldr}$ as our numerical part for definiteness. For the symbolic part we can use any algebriac expression which contains the numerical information[4]. In this chapter we use a square-free (or possibly the minimal) polynomial with integer cofficients as the symbolic part.

For example, $\sqrt{2}$ is an algebraic number since it is a root of x^2-2. Hence our exact representation of this number is

$$[\sqrt{2}, x^2 - 2]$$

where the $\sqrt{2}$ denotes the square root function[5] in $\mathbb{R}_{ldr}$ and the symbolic part, x^2-2, is a defining, in this case minimal, polynomial which has the numerical part as one of

[1]Note $\mathbb{R}_a \subset \mathbb{R}_c \subset \mathbb{R}$.

[2]This contrasts with the situation in most Computer Algebra (CA) systems, where *bigfloats* evaluate to a pre-determined precision. Attempts to produce *lazy bigfloats* in the style of the successful lazy power series have not worked well [7].

[3]In this paper, $\sqrt{\ }$ always means the positive square root.

[4]Typically such an algebraic expression will be a polynomial with coefficients from a ring.

[5]Notice the `Typewrite` font style.

its solutions. The motivation for representing a real number as such a pair comes from an observation that we can use the real root separation bound (calculated exactly), i.e. the minimum distance between any two distinct real roots of a polynomial $p(x) \in \mathbb{Z}[x]$, as a termination condition for the possibly infinite dyadic equality. For the real root separation bound we use the Mahler's [13] separation bound for any two real roots of a given polynomial.

We defined a lazy dyadic equality $=_m$ in Chapter 3. To avoid confusion we will use the symbol $=_p$ ($x <_p y$) for our pair equality (inequality) and reserve the standard equality symbol $=$ ($<$) for mathematical equality (inequality) as in $\sqrt{2} \times \sqrt{3} = \sqrt{6}$.

This chapter is organised as follows.

1. In Section 6.2, we describe `LDRpair`, our exact representation of $\mathbb{R}_a$,
2. In Section 6.3, we define `LDRpair` arithmetic.
3. In Section 6.4, we prove the Equality Theorem and apply it to two examples.
4. In Section 6.5, we prove the Inequality Theorem using Mahler's real root separation bound and show two examples.
5. In Section 6.6, we describe a generic `rootOf` operation, a by-product of our representation.
6. In Conclusion, we discuss various points related with the topic.

6.2 LDRpair: an Exact Representation of $\mathbb{R}_a$

We represent an $x \in \mathbb{R}_a$ by a pair of its lazy dyadic real $\mathbf{x} \in \mathbb{R}_{ldr}$ and its corresponding squarefree polynomial $p(x) \in \mathbb{Z}[x]$.

Definition 6.2.1 (`LDRpair`) *A real algebraic number is a pair* $[\mathbf{x}, p(x)]$, *where* $\mathbf{x} \in \mathbb{R}_{ldr}$ *and* $p(x) \in \mathbb{Z}[x]$ *is a squarefree polynomial*[6] *such that* $p(x) = 0$.

6.3 LDRpair Arithmetic

Notation 6.3.1 *We denote the pair representation of* $x \in \mathbb{R}_a$ *by* $\bar{x}$.

For example, an integer k is represented by $\bar{k} = [\mathbf{k}, x - k]$ in our pair representation. The `res` below denotes the resultant operation [13, 32]. The resultant calculations do not necessarily give (minimal or even) square-free polynomials so we will need some kind of refining operation, which we denote by `R`, to take care of the cases where the resulting polynomials are not square-free. In these cases we square-free-decompose them into products of square-free polynomials. In principle, we can deal with these

[6] This is a deliberate choice on our part. Not insisting on square-free polynomials makes root isolation harder (essentially the root isolation has to do square-free decomposition) while insisting on irreducible polynomials can require full factorization, and we may well have badly behaved (generalised Swinnerton-Dyer) polynomials [26].

polynomials by considering their product, i.e. the square-free part of the original, but we decided to simplify the resulting polynomials by choosing the one which contains the numerical part as a root from among the square-free factors. All these matters are taken care of by the R operation (using Lemma 6.4.6).

Definition 6.3.2 $(+, -, \times, /)$

$$\begin{aligned}
\bar{x} + \bar{y} &:= [\mathtt{x+y}, \mathtt{R}(\mathtt{res}(\mathtt{res}(z-(x+y), p(x), x), q(y), y))] \\
\bar{x} - \bar{y} &:= [\mathtt{x-y}, \mathtt{R}(\mathtt{res}(\mathtt{res}(z-(x-y), p(x), x), q(y), y))] \\
\bar{x} \times \bar{y} &:= [\mathtt{x} \times \mathtt{y}, \mathtt{R}(\mathtt{res}(\mathtt{res}(z-(x \times y), p(x), x), q(y), y))] \\
\bar{x} \;/\; \bar{y} &:= [\mathtt{x\ /\ y}, \mathtt{R}(\mathtt{res}(\mathtt{res}((y \times z)-x), p(x), x), q(y), y))].
\end{aligned}$$

Below we show a simple example of $\bar{2} \oplus \bar{3}$ in Axiom. $\bar{2}$ is $[2, x-2]$ and $\bar{3}$ is $[3, y-3]$[7]. The ? is the Axiom's symbol for a variable[8].

```
(1) -> a := 2::PAIR
   (1)  ["+2.00000",? - 2]                Type: PAIR
(2) -> b := 3::PAIR
   (2)  ["+3.00000",? - 3]                Type: PAIR
(3) -> a + b
   (3)  ["+5.00000",? - 5]                Type: PAIR
```

6.4 An Equality for LDRpair

Recall the defintion lazy dyadic equality, $\mathtt{x} =_n \mathtt{y}$ in Chapter 3, to say that x and y are equal up to n digits. Let $\sqrt{\mathtt{k}}$ be a square root function in $\mathbb{R}_{ldr}$, i.e.

$$\left|\sqrt{\mathtt{k}}(n) - 2^n\sqrt{k}\right| \;<\; 1$$

for positive integer k. For example, $\sqrt{2} \times \sqrt{3} = \sqrt{6}$ becomes

$$[\sqrt{2}, x^2-2] \times [\sqrt{3}, x^2-3] \;=_p\; [\sqrt{6}, x^2-6]$$

in $\mathbb{R}_{pair}$ and we are claiming that the pair equality is the same as the mathematical equality. So we have to show that they are equal in LDRpair.

To derive the Equality Theorem, we need the following theorem of Mahler [13]. Throughout this section $p(x) = a_n x^n + a_{n-1}x^{n-1} + \cdots + a_0$ where the a_i are integers and a_n is non-zero. n is therefore the degree of p. The separation of p, $sep(p)$ is defined as the smallest distance between any two roots of p. Let the roots of p be $\alpha_1, \ldots, \alpha_n$.

Definition 6.4.1 (sep(p))

$$sep(p) \;:=\; \min_{1 \le i < j \le n} |\alpha_i - \alpha_j| \,.$$

[7] In Axiom's language, 2::PAIR and 3::PAIR gives $\bar{2}$ and $\bar{3}$ respectively. We only show five decimal digits of the numerical value for convenience.

[8] In Definition 6.3.2, x's polynomial was in terms of x, y's in terms of y, and the answer's in terms of z. In the implementation, we use Axiom's anonymous "SparseUnivariatePolynomial" type.

$sep(p) \neq 0$ if, and only if, p is square-free.

We also need the concept of the *discriminant* of a polynomial.

Definition 6.4.2 (disc(p)) *The discriminant of a polynomial p, $disc(p)$, with leading coefficient a_n and roots $\alpha_1, \ldots, \alpha_n$ is defined as*

$$disc(p) \quad := \quad a_n^{2n-2} \prod_{1 \leq i < j < n} (\alpha_i - \alpha_j)^2.$$

If a_i are all integers then the discriminant is also an integer, nonzero if, and only if, the polynomial is square-free. Mahler gave a bound for the separation [13].

Theorem 6.4.3 (Mahler)

$$sep(p) > \sqrt{3\,|disc(p)|}\; n^{-(n+2)/2}\; ||p||_2^{1-n}$$

where

$$||p||_2 \quad := \quad \sqrt{\sum_{i=0}^{n} |a_i|^2}.$$

Theorem 4.3 gives us a lower bound for the minimum distance between any two distinct real roots. We will call it Mahler's bound or in short *Mbound.*

Definition 6.4.4 (Mbound(p))

$$Mbound(p) \quad := \quad \sqrt{3\,|disc(p)|}\; n^{-(n+2)/2}\; ||p||_2^{1-n}$$

Using the Mahler's bound we can derive our key theorem.

Theorem 6.4.5 (Equality Theorem) *Let $\bar{x} = [\mathrm{x}, p(x)]$ and $\bar{y} = [\mathrm{y}, q(y)]$. Then*

$$\bar{x} =_p \bar{y} \quad \textit{iff} \quad |\mathrm{x} - \mathrm{y}| < Mbound(r(z))$$

where $r(z)$ is the square-free part of the symbolic part of $\bar{x} - \bar{y}$.

Proof Simply negate Mahler's Theorem. We have

$$|\mathrm{x} - \mathrm{y}| < Mbound(r(z)) \quad \Rightarrow \quad \bar{x} =_p \bar{y}.$$

Note that the refinement operation guarantees that $r(z)$ is a square-free polynomial, i.e. $disc(r(z)) \neq 0$. In the rest of this section we apply our Equality Theorem to two examples.

6.4.1 Example 1 : $\sqrt{2} \times \sqrt{3} = \sqrt{6}$

We apply the Equality Theorem to show $\sqrt{2} \times \sqrt{3} = \sqrt{6}$. In other words we have to show whether

$$[\sqrt{2}, x^2 - 2] \times [\sqrt{3}, x^2 - 3] \quad =_p \quad [\sqrt{6}, x^2 - 6].$$

Expanding the left-hand side we get

$$\begin{aligned} [\sqrt{2}, x^2 - 2] \times [\sqrt{3}, x^2 - 3] &=_p [\sqrt{2} \times \sqrt{3}, \mathtt{R}(\mathtt{res}(\mathtt{res}(z - (x \times y), p(x), x), q(y), y))] \\ &=_p [\sqrt{2} \times \sqrt{3}, \mathtt{R}(x^4 - 12x^2 + 36)] \\ &=_p [\sqrt{2} \times \sqrt{3}, x^2 - 6]. \end{aligned}$$

In Axiom,

```
(1) -> x := sqrt(2::PAIR) * sqrt(3::PAIR)
                      2
   (1)  ["+2.44949",?  - 6]                Type: PAIR
(2) -> y := sqrt(6::PAIR)
                      2
   (2)  ["+2.44949",?  - 6]                Type: PAIR
```

So our problem is now changed to

$$[\sqrt{2} \times \sqrt{3}, x^2 - 6] \quad =_p \quad [\sqrt{6}, x^2 - 6]$$

At this stage we might want to say that they are equal as pairs (hence mathematically equal). But unfortunately we can not yet insist that they are equal as pairs for two reasons:

1. the equality of the symbolic parts is not sufficient for the equality of numerical parts (for example, $\sqrt{2} \times \sqrt{3} \neq -\sqrt{6}$) and
2. the numerical parts are not mathematically equal but lazy dyadically equal.

So we use the Equality Theorem. Subtracting one from the other we have

$$[\sqrt{2} \times \sqrt{3} - \sqrt{6}, x^2(x^2 - 24)].$$

In Axiom (using a version without the refinement operation),

```
(3) -> x-y
                    2
   (3)  ["+0",?(?  - 24)]                  Type: PAIR
```

The polynomial $x(x^2 - 24)$ has three real roots $0, 2\sqrt{6}$ and $-2\sqrt{6}$. Now by the Equality Theorem if

$$|\sqrt{2} \times \sqrt{3} - \sqrt{6}| \quad < \quad Mbound(x(x^2 - 24))$$

then they are equal. Since $Mbound(x(x^2 - 24)) =_5 0.17662$ and $|\sqrt{2} \times \sqrt{3} - \sqrt{6}| =_5 0.00000$ we have shown that they are indeed equal.

6.4.2 Example 2 : $\sqrt{9+4\sqrt{2}} = 1+2\sqrt{2}$

This example is taken from [14]. It corresponds to the pair equality

$$[\sqrt{9+4\sqrt{2}}, x^4 - 18x^2 + 49] \quad =_p \quad [1+2\sqrt{2}, x^2 - 2x - 7].$$

In Axiom,

```
(1) -> x := sqrt(9+4*sqrt(2::PAIR))
                      4      2
   (1)  ["+3.82843",?  - 18?  + 49]      Type: PAIR
```

A factorising refinement would give

```
(2) -> x := sqrt(9+4*sqrt(2::PAIR))
                     2            2
   (2)  ["+3.82843",(?  - 2? - 7)(?  + 2? - 7)]
                                        Type: PAIR
```

and

```
(3) -> y := 1+2*sqrt(2::PAIR)
                     2
   (3)  ["+3.82843",?  - 2? - 7]        Type: PAIR
```

We now have an interesting problem : which one, $x^2 - 2x - 7$ or $x^2 + 2x - 7$, has $\sqrt{9+4\sqrt{2}}$ as one of its roots? This question is not so straightforward to answer. Our answer is to use the lazy dyadic inequality as described below. First we evaluate the two factors at $x = \sqrt{9+4\sqrt{2}}$. Only one of these two evaluations must return zero since one of them must have the given number as a root and all the roots are distinct. Now we **only have to check which evaluation becomes zero.** But this looks like we ended up at the same problem we first set out to solve. But fortunately we have the following lemma. This lemma will allow us to choose the right one among the factors. In this example we have only two factors but the general case is no harder. We will write **defpoly**(x) for the square-free defining polynomial for x.

Lemma 6.4.6 (Refinement Lemma) *Let $p(x)q(x)$ be a factored square-free polynomial corresponding to a real algebraic number x. Hence p, q are square-free and their gcd is* 1. *Then*

$$defpoly(x) = \begin{cases} p(x) & \text{if } |p(\mathbf{x})| < |q(\mathbf{x})| \\ q(x) & \text{otherwise} \end{cases}$$

Proof We know that only one of them is (exactly) zero hence it must be dyadically zero and it is the smaller one (or smallest if there are more than two candidates) and thus it must be the dyadically smaller one.
To apply the lemma, we simply need to check the inequality

$$\left|(x^2 - 2x - 7)_{x=\sqrt{9+4\sqrt{2}}}\right| \quad < \quad \left|(x^2 + 2x - 7)_{x=1+2\sqrt{2}}\right|$$

The actual evalation shows 0.00000 for $x^2 - 2x - 7$ and 15.31371 for the other. So $x^2 - 2x - 7$ is the defining polynomial for $\sqrt{9 + 4\sqrt{2}}$. Having settled the choice problem we now have to settle the equality problem

$$[\sqrt{9 + 4\sqrt{2}}, x^2 - 2x - 7] \quad =_p \quad [1 + 2\sqrt{2}, x^2 - 2x - 7]$$

From now on it is exactly the same routine as the first example. So taking subtraction we get

$$[\sqrt{9 + 4\sqrt{2}} - 1 + 2\sqrt{2}, x(x^2 - 32)].$$

$\sqrt{9 + 4\sqrt{2}} - 1 + 2\sqrt{2} =_5 0.00000$ is less than $Mbound(x(x^2 - 32)) =_5 0.03925$.

6.5 An Inequality for `LDRpair`

In this section we study inequality. We can solve this problem step by step. We use the symbol $<_p$ for pair inequality. First note that we can assert that $\bar{x} <_p \bar{y} = [\mathbf{x}, p(x)] <_p [\mathbf{y}, q(y)]$ (pair inequality) if $\mathbf{x} < \mathbf{y}$. But unfortunately we have no guarantee for the termination of the lazy inequality. Indeed they will run forever if they happen to be equal. Thus what we need is a termination condition which will guarantee the pair inequality without resorting to lazy inequality infinitely[9]. Such a condition can be derived using the pair equality. First, we know that if $\bar{x} =_p \bar{y}$ then obviously $\bar{x} \not<_p \bar{y}$. If not, then we can safely resort to lazy inequality since we are certain that, although this might take arbitrarily long time, they will return either yes or no eventually. Hence we have the following theorem.

Theorem 6.5.1 (Inequality Theorem) *Let* $\bar{x} =_p [\mathbf{x}, p(x)]$ *and* $\bar{y} =_p [\mathbf{y}, q(y)]$. *Then*

$$\begin{aligned}
\bar{x} <_p \bar{y} \quad &\textit{iff} \quad \bar{x} \neq_p \bar{y} \textit{ and } \mathbf{x} < \mathbf{y} \\
\bar{x} >_p \bar{y} \quad &\textit{iff} \quad \bar{x} \neq_p \bar{y} \textit{ and } \mathbf{y} < \mathbf{x} \\
\bar{x} \leq_p \bar{y} \quad &\textit{iff} \quad \bar{x} =_p \bar{y} \textit{ or } \bar{x} <_p \bar{y}, \\
\bar{x} \geq_p \bar{y} \quad &\textit{iff} \quad \bar{x} =_p \bar{y} \textit{ or } \bar{x} >_p \bar{y}.
\end{aligned}$$

6.6 A generic `rootOf` Operation for `LDRpair`

The numeric approach for exact real arithmetic normally provide only algorithms for the square root and the n-th root[10]. In this section we describe a method for computing general radical expressions using $\mathbb{R}_{ldr}$. In the previous section, we used numerical information for deciding the equality. Here we use symbolic information to aid numerical processing. The rationale is that we can replace the numerical part of an $\bar{x}$ by a numerical algorithm directly approximating it as the corresponding root of the

[9]Notice the symmetry: we used inequality information in showing equality. To show inequality we use equality information.

[10]except the work reported in [?].

symbolic part. For example, the numerical part 3.14627... of $\sqrt{2}+\sqrt{3}$, is the same as `rootOf`$(x^4 - 10x^2 + 1, 3)$ assuming that we have such operation. We believe that we need this generic root operation for following reasons:

- (importance) As Abel showed, we can not solve polynomial equations of order bigger than 4 in terms of radicals. For example, `rootOf`$(x^5 + x + 1, -1)$ can't be done any other way.
- (simplicity) It is often troublesome to input a complicated radical expressions. As a simple example, `rootOf`$(x^2 - 210,14)$ is much simpler to type than `(sqrt(2)*sqrt(3)*sqrt(5)*sqrt(7))$PAIR`.
- (efficiency) For those radical expressions where its defining polynomial's total degree and the size of coefficients are small, it is much faster than the lazy dyadic approach, although we have to be more precise what we mean by *those radical expressions where its defining polynomial's degree and the number of coefficients are small.*

We can implement `rootOf` operation in two ways: one using bisection + Sturm sequences and the other using Newton's iteration. Here we describe the version using the Newton's method. Note that one of the key problem in applying Newton's iteration (in numerical mathmatics) is to find the initial starting point. But we don't have this difficulty thanks to our pair representation. Basing on the observation that 3.14627... is the same as `rootOf`$(x^4 - 10x^2 + 1, 3)$ we chose the type of `rootOf` as $(\mathbb{Z}[x], \mathbb{R}_{ldr}) \to \mathbb{R}_{ldr}$. The algorithm itself is quite simple. Given a pair of polynomial $p(x)$ and an approximation $\mathtt{y}$, $(p(x), \mathtt{y})$, the `rootOf` operation

1. checks whether the $p(x)$ has one, and only one, root in the interval given by the second argument, i.e., $(\mathtt{y}(0) - e, \mathtt{y}(0) + e)$, where e is the allowed error bound which we can choose, say 1. If no, then report error, otherwise proceed.
2. Newton-iterate with
   - starting value: $\mathtt{y}(0)$.
   - stop condition: $|(\mathtt{y}(0) - \frac{p(\mathtt{y}(0))}{p'(\mathtt{y}(0))}) - \mathtt{y}(0)| < \frac{1}{2^n}$.
3. returns $\lfloor 2^n \times$ (the result of step 2)$\rceil$.

6.6.1 Finite Product of Simple Radicals

Here we are interested in finite products of simple radicals, i.e., $\prod_{i=0}^{m-1} \sqrt{\mathtt{k}_i}$ (case 1), $\prod_{i=0}^{m-1} \sqrt[n]{\mathtt{k}_i}$ (case 2) or combination of these two.

If we evaluate, say $\sqrt{2} * \sqrt{3} * \sqrt{5} * \sqrt{7}$, using the usual $\sqrt{}$ operation of the lazy dyadic arithmetic, then we need to perform four (lazy dyadic) multiplications, which is costly, and also we may have to evaluate up to twice the given precisions for each of the `sqrt` operation.

- (case 1) $\prod_{i=0}^{m-1}\sqrt{k_i}$, where for each i, k_i are positive integers. In this case the polynomial is of the form $x^2 - c$ where $c = \prod_{i=0}^{m-1} k_i$. In this case, `rootOf` is much faster than the lazy dyadic approach.
- (case 2) $\prod_{i=0}^{m-1}\sqrt[n]{k_i}$, where $n > 2$ and for each i, k_i are positive integers. Again `rootOf` performs much better than the lazy dyadic approach.

6.6.2 Finite Summation of Simple Radicals

This is a thorny case and our `rootOf` operation seems worse than the ordinary combination of the square root and n-th root. The main reason for this is that the resulting polynomial corresponding to the the numerical part usually has a sizable total degree and also has several non-zero coefficients. In these cases it seems that the original lazy dyadic approach is much faster than our `rootOf`. Below is an example of Axiom session evaluating $\sqrt{2}+\sqrt{3}+\sqrt{5}+\sqrt{7}$. Notice the enormous time difference between the two approaches.

- (Lazy Dyadic Approach)

```
(1) -> (sqrt(2)+sqrt(3)+sqrt(5)+sqrt(7))$LDR

   (1)  "+8.02808"                                    Type: LDR
```

  Evalation (in lazy dyadic approach) of the same expression to many precisions, say hundreds, does not cost much extra.

- (Pair Approach) To see what kind of polynomial we are talking about we evalaute the same expression in pair model.

```
(1) -> (sqrt(2)+sqrt(3)+sqrt(5)+sqrt(7))$PAIR

   (1)
   ["+8.02808",

        16       14        12          10          8
      ?   - 136?   + 6476?   - 141912?   + 1513334?
                6           4           2
      - 7453176?  + 13950764?  - 5596840?   + 46225]

                                                 Type: PAIR
    Time: 0.07 (IN) + 0.08 (EV) + 0.05 (OT) = 0.20 sec
```

  The polynomial is of total degree 16 and dense. Also the coefficients are quite large. This tends to make the intermidiate rational numbers huge in the Newton iteration. How to make intermediate calculations less is a topic for future research.

```
(2) -> rootOf(sp %% 28,8::LDR)

   (2)  "+8.02808"
                                                        Type: LDR
        Time: 0.07 (EV) + 174.90 (OT) + 0.64 (GC) = 175.61 sec
```

Chapter 7

Implementations in Axiom

7.1 Introduction

Having constructed a category `LazyReal` in Axiom for $\mathbb{R}_c$ in Chapter 5, the next task is to implement the two models of $\mathbb{R}_c$, $\mathbb{R}_{ldr}$ and $\mathbb{R}_{lft}$, as two domains in the category `LazyReal`. The result of this *implementing as a domain* is shown in Chapter 5, Section 4. So here we describe various issues in implementing some important functions in both domains.

7.2 $\mathbb{R}_{ldr}$ as a domain of LazyReal

Our implementation of $\mathbb{R}_{ldr}$ in Axiom is based on Harrison's implementation [18, 19] of $\mathbb{R}_{ldr}$ in Camllight. Unfortunately Ménissier-Morain's implementation of $\mathbb{R}_{ldr}$ in Caml[1] was not available to refer to. Most of the implementation in Axiom was a straightforward translation of the one in Camllight except transcendental functions. We have already mentioned some important functions, `ndiv`, `separate`, `memo` and `msd`, of $\mathbb{R}_{ldr}$ in Chapter 3.

7.2.1 basic arithmetic operations

The four basic operations, $+, -, \times, \div$, are rather straightforward to define. Addition and subtraction is quite simple.

```
(x:%)+(y:%):% == memo(per((n:I):I +->
                  ndiv(nthint(x,n+2)+nthint(y,n+2), 4)));
-(x:%):% == memo(per((n:I):I +-> -nthint(x,n)));
(x:%)-(y:%):% == x+(-y);
```

where the `nthint` function, given an $x \in \mathbb{R}_{ldr}$, returns $x(n)$, the value of x at n. Note that we could have defined the subtraction directly without defining the unary minus as below.

[1] Caml is a functional language developed by INRIA

```
(x:%)-(y:%):% == memo(per((n:I):I +->
                    ndiv(nthint(x,n+2)-nthint(y,n+2), 4)));
```

This is possible since our categories are not based on Axiom's built-in categories (Axiom's category `AbelianGroup` defines subtraction as we did above) and this might be a better implementation in terms of performance.

One contrast between other languages and Axiom is that in Axiom we can use the same symbol for many operations, i.e., symbol overloading, whereas in other languages such as camllight and Caml we couldn't. For example, in Axiom, we can use $\star$ for the following many multiplications.

```
*:      (I,%)->%;
*:      (%,I)->%;
*:      (Q,%)->%;
*:      (%,Q)->%;
*:      (%,%)->%;
```

where, as before, the % means an element of $\mathbb{R}_{idr}$. The actual coding for the above multiplications is as below.

```
(x:%)*(k:I):% == {
        memo(per((n:I):I +-> {
                p:I := logtwo(abs(k));
                p1:I := p+1;
                m:I := nthint(x,n+p1);
                ndiv(k*m,pown(p1))
                }
        ))
}
(k:I)*(x:%):% == x*k;
(x:%)*(r:Q):% == (x*numer(r)) / denom(r);
(r:Q)*(x:%):% == x*r;
(x:%)*(y:%):% == {
        memo(per((n:I):I +-> {
                n2:I := n+2;
                r:I := n2 quo 2;
                s:I := n2 - r;
                xr:I := nthint(x,r);
                ys:I := nthint(x,s);
                p:I := logtwo(xr);
                q:I := logtwo(ys);
                zero?(p) and zero?(q) => 0;
                k:I := q+r+1;
                l:I := p+s+1;
                m:I := p+q+4;
                xk:I := nthint(x,k);
                yl:I := nthint(y,l);
```

```
                ndiv(xk*yl, pown(m))
                }
        ))
}
```

The `logtwo` function is a binary logarithm operation on integers, i.e., given an integer m it finds the smallest p such that $2^p \geq |m|$. `pown`(k) means raising 2 to the power k. Having operations for multiplication by an integer or a rational number makes the implementation more efficient. The same applies to division and we can define division by an integer or a rational number separately.

```
(x:%)/(k:I):% == memo(per((n:I):I +-> ndiv(nthint(x,n),k)));
(x:%)/(r:Q):% == (x*denom(r))/numer(r);
```

Using these we can define multiplicative inverse for an integer or a rational separately as we did for multiplication.

```
inv(k:I):% == (1$%)/k;
inv(r:Q):% == (denom(r)::%)/numer(r);
inv(x:%):% == {
        memo(per((n:I):I +-> {
                x0:I := nthint(x,0);
                k:I  := {
                        x0 > 1 => {
                                r:I := logtwo(x0)-1;
                                m:I := (n+1)-mul2(r);
                                m < 0 => 0;
                                m
                        }
                        p:I := msd x;
                        mul2(n)*(p+1);
                }
                xk:I := nthint(x,k);
                ndiv(pown(n+k),xk)
        }))
}
```

where the `mul2`(k) means simply k multiplied by 2. Finally we can define a division

```
(x:%)/(y:%):% == x * inv(y);
```

7.2.2 ordering

The fundamental concept of separating a computable real number from zero or separating two computable real numbers are defined in Chapter 3. The actual recursive definitions of these functions are given below. First, the function `sep` which separates a given computable real number from 0 and the `sign` function using it.

```
sep(x:%):Z == {
        sepAux(n:Z,x:%) : Z == {
                local d:Z;
                d := nthint(x,n);
                abs(d) > 0 => d;
                sepAux(n+1,x)
        }
        sepAux(0,x)
}
sign(x:%):Z == {
        sep(x) > 0 => 1;
        sep(x) < 0 => -1;
        0
}
```

Then the `separate` function and ordering relations.

```
separate(x:%, y:%):Z == {
        separateAux(n:Z,x:%,y:%) : Z == {
                local d:Z;
                d := nthint(x,n) - nthint(y,n);
                abs(d) > 1 => d;
                separateAux(n+1,x,y)
        }
        separateAux(0,x,y)
}
(x:%)>(y:%):Boolean == separate (x, y) > 0;
(x:%)<(y:%):Boolean == separate (x, y) < 0;
(x:%)<=(y:%):Boolean == not (x < y);
(x:%)>=(y:%):Boolean == not (y > x);
```

Note that we could have defined the $<$ relation using $>$ as below.

```
(x:%)<(y:%):Boolean == y > x;
```

7.2.3 transcendental functions

The four main transcendental functions are described in Chapter 3 and in Appendix B. Here we will only describe `sqrt` and `nthRoot` functions. Definitions of these functions are based on [34]. The `nthRoot` function uses the Axiom's built-in `approxNthRoot` operation. For a given integer n and a non-negative integer r, `approxNthRoot`(n,r) returns an integer approximation k to $\sqrt[r]{n}$ such that $|k - \sqrt[r]{n}| < 1$.

```
nthRoot(x:%,k:NI):% == {
        memo(per((n:Z):Z +-> {
                local xkn:Z;
                xkn := nthint(x,k*n);
```

```
                    zero?(xkn) or xkn > 0 =>
                            approxNthRoot(xkn, k);
                    error "nthRoot"
                    }))
    }

    sqrt(x:%):% == nthRoot(x,2);
```

7.3 $\mathbb{R}_{lft}$ as a domain of LazyReal

We couldn't implement $\mathbb{R}_{lft}$ in Axiom mainly due to lack of time. Also, the mutually recursive defintion of domains in Axiom seems quite a daunting task. So here we only describe how this can be done.

There are several implementations of $\mathbb{R}_{lft}$ in various languages which one can try to translate into Axiom: in Miranda (Potts), in Caml (Potts), and in C (Lindsay Errington et. al.). The Miranda implementation doesn't look like a good translation candidate mainly due to difference between Miranda's treatment of lazy types and that of Axiom. The Caml implementation, called **Calathea** by Potts, uses Caml's elegant implementation of arbitrary rational numbers. Our description below is based on Calathea. Calathea defines basic types such as *vector, matrix, tensor* and types such as *sign, domain, lft.* But the key types such as *uefp* (unsigned exact floating point), *ureal* (unsigned real), *sefp* (signed exact floating point), *sreal* (signed real), *uexp* (unsigned expression tree), *sexp* (signed expression tree) are defined in mutually recursive. As said above this is not straightforward to do in Axiom.

We can define the basic types *vector, matrix* and *tensor* as domains `LV`, `LM` and `LT` respectively as record types.

```
LV == Record(num1:Z,num2:Z)
LM == Record(vec1:LV,vec2:LV)
LT == Record(mat1:LM,mat2:LM)
```

We can define `LD` and `LFT`, corresponding to *domain* and *lft* respectively, as union types

```
LD == Union(bottom:'bottom',total:LV,partial:LM)
LFT == Union(vector:LV,matrix:LM,tensor:LT)
```

We can define a package `LE` which contains several function related with the informtaion overlap strategy. The domain `LS` defines the four signs, `POS, NEG, ZER` and `INF`, as enumeration types.

```
LS == Enumeration(POS:%,INF:%,NEG:%,ZER:%)
```

The following is a part of Potts' definition of the key types such as *ureal* and *sreal* in Caml.

```
type    sreal == sefp ref and
        ureal == uefp ref and
```

```
sefp == sign * ureal and
uefp = Unterm of digits * ugen | Term of digits * vec and
sexp = Svec of vec |
       Smat of mat * uarg |
       Sten of ten * uarg * uarg |
       Sclos of sclos and
uexp = Uvec of vec |
       Umat of mat * uarg |
       Uten of ten * uarg * uarg |
       Uclos of uclos and
uarg = Udesc of ureal * int | Uexp of uexp and
ugen == num -> ureal and
sclos == num -> sexp and
uclos == num -> uexp;;
```

As we can see from above the types are mutually recursive. To illustrate how to do this sort of thing in Axiom we isolate the three types (*ureal*, *uefp* and *ugen*) since other types depends on these.

```
type ureal == uefp ref and
     uefp ==  Unterm of digits * ugen | Term of digits * vec and
     ugen == num -> ureal;
```

Observe that *ureal* depends on *uefp* and *uefp* depends on *ugen* and *ugen* depends on *ureal*. What we want is three domains in Axiom, say `Ureal`, `Uefp` and `Ugen`, implementing the above three types. But in Axiom we cannot define several domains at one time. One solution is to write categories for the domains and break the recursion oneself[2]. So we write three categories `UrealSpec`, `UefpSpec`, and `UgenSpec` which export some functions required by other domains. For instance,

- `UrealSpec` exports funReal which is used in `Uefp` or `Ugen`,
- `UgenSpec` exports funGen which is used in `Ureal` or `Ugen`, it should be a parametrized category with parameter `Uabstract:UrealSpec`, and
- `UefpSpec` exports funUefp which is used in `Ureal`, it should receive two parameters `Uabstract:UrealSpec` and `Umake:UgenSpec(Uabstract)`. Thus Umake is a dependent type.

Now we can define `Ugen` as a parametrized domain `Ugen(Uabstract:UrealSpec)` and `Uefp` as `Uefp(Uabstract:UrealSpec,Umake:UgenSpec(Uabstract))`. And finally we can define `Ureal` as

```
TheUgen == Ugen(%)
TheUefp == Uefp(%,TheUgen(%))
Ureal == Reference(TheUefp)
```

[2]Thanks to Renaud Rioboo for this advice.

7.4 $\mathbb{R}_{LDRpair}$ as a domain of Pair

As we briefly mentioned in Chapter 5 we defined LDRpair as a domain of the Pair category. The only difference between LDRpair and LFTpair was in the representation of the numerical part. We represented LDRpair as a record type as below.

```
LDRpair == Record(num:LDR, sym:SUP(I))
```

where *num* is of type LDR and *sym* is of type SUP(I) (recall that SUP(I) abbreviates SparseUnivariatePolynomial(Integer)).

7.4.1 arithmetic operations

Hence, in LDRpair, a rational number r is represented as a record with its numerical part is r as an LDR and its symbolic part is the defining polynomial of the rational number. For example, 1/3 is represented as

```
["+0.33333",3? - 1]
                                        Type: LDRpair
```

The four arithmetic operations, $+, -, \times, \div$, are defined in Chapter 6 where we used LDRpair to describe the equality algorithm and a generic root operation. As an example, addition was defined as below

$$\bar{x} + \bar{y} \;:=\; [\mathtt{x} + \mathtt{y}, \mathtt{R}(\mathtt{res}(\mathtt{res}(z - (x + y), p(x), x), q(y), y))]$$

In Axiom the above definiton becomes the following.

```
(a:%)+(b:%):% == {      -- z-(x+y) ==> -x-y+z
        x:LDR := np a;
        y:LDR := np b;
        p:SUP I := sp a;
        q:SUP I := sp b;
        s:SUP I := monomial(1,1); -- this is z
        t:SUP SUP I:= monomial(-1,1)+monomial(s,0); -- this is -y+z
        u:SUP SUP SUP I := monomial(-1,1)+monomial(t,0);
                -- this is -x-y+z
        r1:SUP SUP I := resultant(u,lift2 p);
        r2:SUP I := resultant(r1,lift1 q);
        refine(per [x+y, r2])
}
```

In the definition above, $np(x)$ and $sp(x)$ returns the numerical part and symbolic part of x respectively. There are several *monomial* operations used above[3]: s corresponds to z, t corresponds to $z-y$ and u corresponds to $z-(x+y)$. The *lift1* operation lifts an SUP(I) to SUP(SUP(I)) and the *lift2* operation lifts an SUP(I) to SUP(SUP(SUP(I))). The *resultant* operation calculates the resultant of the given two polynomials.

[3]The *monomial* operation is the basic building block to form a polynomial in Axiom

7.4.2 the refine operation

The resultant calculations do not necessarily give (minimal or even) square-free polynomials so we will need some kind of refining operation, which we denote by R, to take care of the cases where the resulting polynomials are not square-free. In these cases we square-free-decompose them into products of square-free polynomials. In principle, we can deal with these polynomials by considering their product, i.e. the square-free part of the original, but we decided to simplify the resulting polynomials by choosing the one which contains the numerical part as a root from among the square-free factors. All these matters are taken care of by the refine operation and its code is as below.

```
refine(x:%):% == {
        numpart:LDR := np x;
        sympart:SUP I := sp x;
        f:Factored SUP I:= squareFree(sympart);
        numberOfFactors f = 1 =>
                per [numpart, apply(factorList(f).1,fctr)];
        lift(p:SUP I): SUP Q == map((m:I):Q +->
                coerce(m)$Q, p)$UPCF(I,SUP I,Q,SUP Q);
        factorlist:List SUP I :=
                [apply(r,fctr) for r in factorList(f)];
        sturmed:List Record(pol:SUP I, st:List SUP Q) :=
                [[p,[lift(pp) for pp in sturm p]] for p in factorlist];
        interval(n:I):Record(left:Q,right:Q) ==
                [(nthint(numpart,n)-1)/2^(n::NI),
                 (nthint(numpart,n)+1)/2^(n::NI)];
        svzero(r:Record(left:Q,right:Q), l:List SUP Q):Boolean
                == {
                l1:List Q := [apply(p,r.left) for p in l];
                l2:List Q := [apply(p,r.right) for p in l];
                sv:I := variation(l1)-variation(l2);
                sv = 0
        }
        sturmed:=[u for u in sturmed | not svzero(interval(0),u.st)];
        precision:I := 0;
        while (#sturmed>1) repeat {
                precision := precision+1;
                sturmed:=[u for u in sturmed |
                        not svzero(interval(precision),u.st)];
                }
        per[numpart,sturmed.1.pol]
}
```

UPCF denotes the Axiom's constructor UnivariatePolynomialCategoryFunctions.

Chapter 8

Performance Comparisons

8.1 Introduction

One of the main advantages in implementing various models of exact real arithmetic in Axiom as domains of the same category `LazyReal` ($\mathbb{R}_c^{num}$) is that we have a uniform platform on which we can compare the two representations fairly. Indeed this was one of the motivations for choosing Axiom since its powerful algebraic structure allows us to do that. Unfortunately we didn't have time to implement `LFT` to campare with `LDR`. Instead we decided to compare both within `LDR` and between `LDR` and variants of `LDR`s. Note that in `LDR` we used base 2. We implemented two other variants by choosing different bases, namely `LQR` (base 4), and `LHR` (base 16). A computable real number x is represented by a recursive sequence `x` such that

$$|\mathtt{x}(n) - 4^n x| < 1$$

in `LQR` and by a recursive sequence `x` such that

$$|\mathtt{x}(n) - 16^n x| < 1$$

In this chapter all comparisons are done on a Sun Sparc machine and timings are measured by Axiom's built-in timer.

8.2 Comparisons within `LDR`

8.2.1 finite summation

Recall that we have to evaluate two more precisions to add two numbers in `LDR`. For example, to get n exact binary digits of `x + y` in `LDR` we have to evaluate both `x` and `y` up to $n + 2$ binary digits. Here we want 10 binary digits of two finite summation expressions:

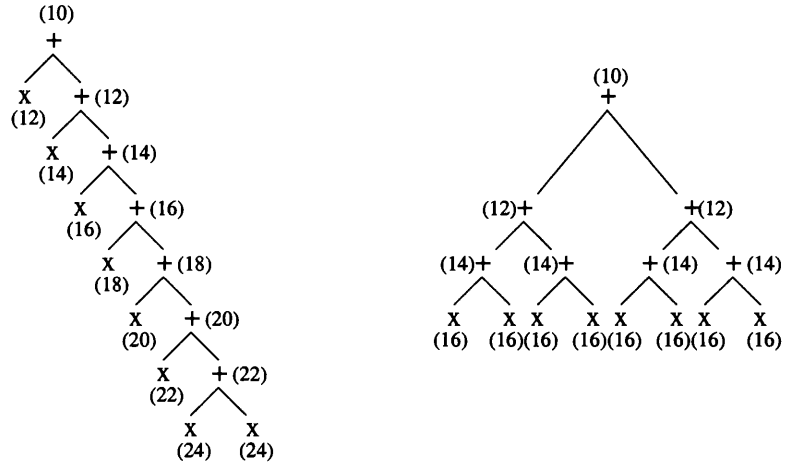
```
x+(x+(x+(x+(x+(x+(x+(x)))))))   (Expression 1)
((x+x)+(x+x))+((x+x)+(x+x))     (Expression 2).
```

Table 8.1: Timings

n	expression 1	expression 2	refined expression 1	refined expression 2
100	0.04	0.05	0.03	0.06
1000	0.26	0.23	0.25	0.49
10000	5.33	5.09	5.20	5.12

For explanation, we let $\mathtt{x} = \frac{1}{3}$. As we can see from the Figure 8.1 evaluation of Expression 2 forms a balanced binary tree while evaluation of Expression 1 forms a skewed binary tree.

Figure 8.1: Evaluation trees of expressions 1 and 2 upto 10 binary digits



As the numbers inside the bracket shows that the skewed expression evaluates $\mathtt{x}$ upto 24 bits, while the balanced expression evaluates only upto 16 bits. Of course we can evaluate Expression 1 using a direct summation as $\sum_{i=0}^{7} \mathtt{x}_i$ as defined in Appendix A. This gives a tighter error bound and only evaluates $\mathtt{x}$ upto 14 bits. But we can do the same to Expression 2 as $\sum_{i=0}^{3} 2x$. Hence, the balanced addition will give a better performace on both occasions, brute force or direct summation. Table 8.1 shows the timings of the two expressions and of their direct summation versions for various n. As we can see the balanced binary addition starts to pay off at higher precisions.

8.2.2 finite product

We can do the same kind of comparison for multiplication as we did for addition. Note that to evaluate xy, say $x = y = \frac{1}{3}$, to $n = 10$ binary digits LDR evaluates both x's upto 42 binary digits. Again we evaluate the two expressions:

$$\mathtt{x}\times(\mathtt{x} \times (\mathtt{x} \times (\mathtt{x} \times (\mathtt{x} \times (\mathtt{x} \times (\mathtt{x})))))) \quad \text{(expression 3)}$$
$$((\mathtt{x} \times \mathtt{x}) \times (\mathtt{x} \times \mathtt{x})) \times ((\mathtt{x} \times \mathtt{x}) \times (\mathtt{x} \times \mathtt{x})) \quad \text{(expression 4).}$$

The evaluation trees for both expressions are shown below. Table 8.2 shows the timings of the two expressions for various n.

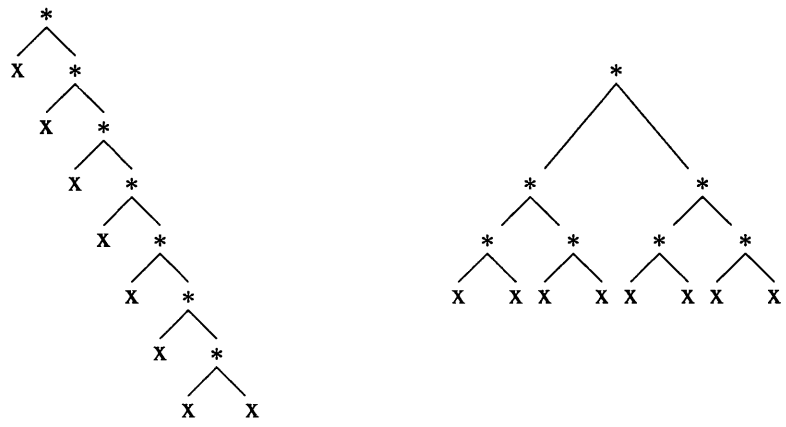


Figure 8.2: Evaluation trees of both expressions upto 10 binary digits

Table 8.2: Timings for the two expressions

n	expression 3	expression 4
100	1.66	2.71
500	3.77	6.48
1000	15.27	16.37

8.3 Comparisons among LDR, LQR and LHR

8.3.1 finite summation

We compare the performances of expression 1 for each $\mathtt{x} \in$ LDR, LQR and LHR in Table 8.3.

8.3.2 finite product

We compare the performances of expression 3 for each $\mathtt{x} \in$ LDR, LQR and LHR in Table 8.4.

8.3.3 $\sqrt{\sqrt{\sqrt{\sqrt{x}}}}$

We compare the times taken to evaluate the expression $\sqrt{\sqrt{\sqrt{\sqrt{\mathtt{x}}}}}$. we let $\mathtt{x}= 2$. The time given is the total time and the time inside the bracket indicates evaluation time.

Table 8.3: Timings for expression 1

n	LDR	LQR	LHR
100	0.04	0.03	0.06
1000	0.24	0.23	0.45
10000	5.12	6.55	5.13

Table 8.4: Timings for expression 1

n	LDR	LQR	LHR
100	0.06	0.05	0.06
1000	0.22	0.27	0.41
10000	5.82	6.95	5.22

Table 8.5: Timings for $\sqrt{\sqrt{\sqrt{\sqrt{2}}}}$

n	LDR	LQR	LHR
100	0.03	0.05	0.08
1000	3.27	3.24	4.95
10000	310.22	311.22	304.0

Note that to evaluate $\sqrt{2}$ to n binary digits our definition of the square root function evaluates 2 up to $2n$ binary digits. Since our expression has nesting degree of 4, this means that it evaluates $2^4(2n) = 2^5 n$ in binary.

Chapter 9

Conclusion and Further Study

9.1 Overview of thesis

Chapter 2 We gave a brief survey on exact real arithmetic: its history, properties and problems.

Chapter 3 We described the lazy dyadic exact real arithmetic which is largely thanks to Hans Boehm [7, 5, 6] and Valerie Ménissier-Morain [34, 33]. Our description is largely based on [18] and we do not claim any originality except some hand proofs for the error bounds for the elementary functions. We needed an integer division with error less than or equal to half at all times and observed that this depends on the rounding mode of the base system. The lazy dyadic model is not incremental and this means that, for each expression, we need to cache the highest precision already evaluated for efficiency. Based on the fundamental concept of digit separation we defined ordering relations for the computable real numbers. Transcendental functions are represented in terms of Taylor series. The main task in implementing these functions was identifying the truncation errors and the summation errors.

Chapter 4 We described the linear fractional exact real arithmetic. Gosper made the fundamental observation that calculation with continued fractions is manageable if we represent them as `lff` [15]. Then, based on Gosper's idea, Vuillemin gave a representation of $\mathbb{R}_c$ as an infinite product of homographies and developed algorithms for algebraic and transcendental operations [47]. Continuing this development, Potts and Edalat gave another representation, called `efp`, which can be thought of as a generalization of Vuillemin's work. Further work on this model has also been done by Heckmann.

Chapter 5 We developed an Axiom [25] category structure for computable real numbers based on the fact that they form a field but this field does not have equality[1]. But they have, what we might call, a lazy equality. Based on this observation we constructed a category, `LazyOrderedField`, for the class of field with lazy equality in Axiom. We had to build this from scratch, but we followed closely Axiom's

[1] For a clear account of the problems of equality in computer algebra one should look at [12]

built-in category for the class of commutative field. We also defined a category, `LazyReal` for the class of numerically-representable computable real numbers as an extended category of `LazyOrderedField`. We then implemented two models of exact real arithmetic as domains, `LDR` and `LFT` respectively of the category `LazyReal`. This can be thought of as the first realization of exact real arithmetics as abstract data types inside a computer algebra system. We also defined a separate category, `Pair` for exact real algebraic arithmetic. By pairing symbolic and exact-numerical representation we obtained `LDRPair` as a domain of `Pair`. Hence this work is a realization of computable real numbers as an abstract data type. We also built a category for real algebraic numbers and represented them as a pair of numerical and symbolic information associated with them. Axiom's powerful category structure was a valuable source of information in determining the suitable category for computable real numbers.

Chapter 6 Using Axiom's algebraic power, we developed an exact model for real algebraic numbers. Real algebraic numbers are special in the sense that we have Mahler's bound for root separation. Using this model we have shown that we can decide the equality and inequality for $\mathbb{R}_a$. But in general, Rice [40] showed that zero is undecidable. Due to this result we can not, for example, determine the integer part of a real number which, in turn, implies the undecidability of the rationality/irrationality of a real number. But now with our algorithm we can at least test equality for any two real algebraic numbers[2]. The key result in this process is Theorem 6.4.5, which relies on the bound in Lemma 6.4.3. If such a bound could be found for a wider class of expressions, this approach would generalise.

We regret that we couldn't perform a complexity analysis of our algorithms. As far as we know the B-adic exact real arithmetic itself lacks any kind of complexity information (the same situation holds for the linear fractional transformation approach). Real algebraic numbers are exactly representable and already there are many representations for them. But as far as we know the representation using exact real numbers as numerical part is the first of its kind. The main difference from other models for algebraic number arithmetic such as [32, 44] is that we use exact real arithmetic whereas others mostly use floating point interval arithmetics. This gives an interesting contrast between our pair representation and those (minimal polynomial, interval) pair representations: their interval arithmetics have to specify *which root* of the polynomial is meant whereas we have to choose *which polynomial* contains the root that is given as the numerical part. This is due to the fact that our model includes the exact numerical information while others include the root-location information in terms of intervals. In choosing the right polynomial we fully used the numerical information provided by lazy

[2]At the ISSAC 2000 Conference at St Andrews, we met Rioboo who kindly explained his work on real closed field. In [42] he developed an Axiom category for real closed fields in which he gave a set of functions for computing generic real roots of polynomials. His demonstration showed that his algorithm is a lot faster than ours. But his approximation is not exact. In the same vein, we can cite [9, 50].

dyadic exact real arithmetic. For other representations of algebraic numbers see those references cited above and [35]. These representations seem to differ only in the way how they locate the roots of the polynomials and all seem to rely on the calculation of Sturm sequences to count the number of sign variations, and thereby prove that the interval contains a unique root.

Chapter 7 We described the implementation of lazy exact real arithmetic and the pair arithmetic in detail. The only difference between $\mathbb{R}_{LDRpair}$ and $\mathbb{R}_{LFTpair}$ is in the representation of the numerical part. The rest is exactly the same.

Chapter 8 We compared the performances of various operations both within the lazy dyadic exact real arithmetic and with other variants. As one can see from the tables given, choosing a different base (4 or 16 rather than 2) doesn't seem to make any difference in performance. We don't have to compare the performances of `LDRPair` and `LFTPair` since this only depends on the performances of the numerical part which we already did above.

9.2 Conclusion

At the core of exact real arithmetic is the concept of computable real numbers. Combining this concept with sequence we get the lazy B-adic representation `LDR`, and with continued fraction we get the linear fractional transformation representation `LFT`.

Axiom's abstract data types such as categories, functors, domains and packages allowed us to develop a coherent structure for $\mathbb{R}_c$. Its computational power has been invaluable to us.

`LDRPair` proves a satisfactory model for a subset of $\mathbb{R}_c$, representing both the traditional numerical view, via the `LDR` representaton, and a symbolic view which lets us treat equality correctly. It relies on the knowledge of the root separation, which is based on Lemma 6.4.3. It is interesting and important to know how far we can push this kind of approach to a larger subset of real numbers.

9.3 Further Research Topics

Below we give a list of topics for further study.

- We didn't look at the problem of representing a computable real *function*. The interested reader should look at the works of Lacombe, Grzegorczyk, Banach, Mazur and Pour-El and Richard [39].
- As pointed out it would be interesting to implement transcendental functions using other methods: Brent's method [8], Chebyshev series, or the Cordic algorithms.
- We were unable to implement Potts and Edalat's model called exact floating point in Axiom due to lack of time and its complexity. A proper comparison of this with `LDR` still remains to be done. It would be nice to have an implementation of `LFT` as a domain of `LazyReal` and `LFTPair` as a domain of `Pair`.

- The next step for the zero recognition problem (equivalent to the equality problem) is to consider the possibility of zero recognition for a larger class of numbers which includes some transcendental numbers. Currently Richardson's method [41] seems the best at the moment. Although Richardson's method can also be applied to transcendental numbers it uses a combination of difficult mathematics such as LLL algorithm [30] and Wu's method for generating characteristic sets and we do not know, for example, its complexity. One important result related to this is the model-completeness proof for the first order theory $(\mathbb{R}, \exp)$ [48].

- Another area related to our work is in denesting nested radicals [29]. It will be interesting if we can incorporate these simplifications into our algorithm so that we can simplify nested radicals first before testing equality.

- Several experiments need to be performed to tune this implementation. For example, it would be possible to replace the numeric part of $\sqrt{2}+\sqrt{3}$, whose minimal polynomial is $x^4 - 10x^2 + 1$, by a numerical algorithm directly approximating this root (we do know a starting value, generally the major problem in numerical root-finding). Is this worth it?

- We have only discussed real arithmetic. Since $\mathbb{C}_a \cong \mathbb{R}_a \times \mathbb{R}_a$, we have an exact representation of $\mathbb{C}_a$ as $\mathbb{R}_{\texttt{LDRpair}} \times \mathbb{R}_{\texttt{LDRpair}}$. But is this the most efficient one? Maybe we should have a triple $(\Re(x) \in \mathbb{R}_{ldr}, \Im(x) \in \mathbb{R}_{ldr}, p)$, where p is a polynomial satisfied by x.

- The generic `rootOf` operation shows much better timing performance in the case of finite products of simple radicals than the lazy dyadic approach. Certainly we need to refine our implementation in several ways. First we may be able to reduce the evaluation precision inside the Newton iteration procedure. Second we can make the implementation more user-friendly by allowing the user to specify an interval in which the root lies. Also, using bisection, instead of Newton iteration, might be useful in splitting the intervals.

- Whilst attending the ISSAC 2000 conference at St. Andrews in Scotland, we became aware of several related works by others, most notably Rioboo's work on real closed field [42]. Rioboo implemented a parameterized domain constructor called `RealClosure` in Axiom. Since the real closure of $\mathbb{Q}$ is $\mathbb{R}_a$, we can declare $\mathbb{R}_a$ as `RealClosure`$(\mathbb{Q})$. The `RealClosure` has all the usual operations associated with $\mathbb{R}_a$ and its equality test is much more efficient than ours. Our current implementation is very slow and we can further improve its efficiency in several places. Rioboo's work on real closed fields is notable for its efficiency [42].

  Let us end with a diversion. We all know that $\mathbb{R}_{ldr}$ is a real closed field. Then

$$\mathbb{R}_{ldr} = \texttt{RealClosure(?)}?$$

Appendix A

Proofs of $\mathbb{R}_{ldr}$ algorithms

We give correctness proofs of the algorithms for various $\mathbb{R}_{ldr}$ operations. Note that almost all of these proofs are taken from [34, 33, 18, 19]. Here, if x denotes a computable real then its typewriter font $\mathtt{x}$, denotes the representation of x in $\mathbb{R}_{ldr}$, i.e., a function of type $\mathbb{N} \mapsto \mathbb{Z}$. Recall $|\mathtt{x}(n) - 2^n x| < 1$ by definition.

Definition A.0.1 (integer k) $\mathtt{k} := n \mapsto 2^n k$

Proof

$$|\mathtt{k}(n) - 2^n k| = |2^n k - 2^n k| = 0 < 1$$

Definition A.0.2 ($-\mathtt{x}$) $-\mathtt{x} := n \mapsto -\mathtt{x}(n)$

Proof

$$\begin{aligned} |-\mathtt{x}(n) - 2^n(-x)| &= |-(\mathtt{x}(n) - 2^n x)| \\ &= |\mathtt{x}(n) - 2^n x| \\ &< 1 \end{aligned}$$

Definition A.0.3 ($|\mathtt{x}|$) $|\mathtt{x}| := n \mapsto |\mathtt{x}(n)|$

Proof

$$\begin{aligned} ||\mathtt{x}|(n) - 2^n|x|| &= ||\mathtt{x}(n)| - 2^n|x|| \\ &= ||\mathtt{x}(n)| - |2^n x|| \\ &\leq |\mathtt{x}(n) - 2^n x| \quad \text{(using } ||x| - |y|| \leq |x - y|) \\ &< 1 \end{aligned}$$

Theorem A.0.4 (memo)

$$\forall n, \forall k \geq 1.\ \left|\mathtt{x}(n+k) - 2^{n+k}x\right| < 1 \ \Rightarrow\ \left|\left\lfloor \mathtt{x}(n+k)/2^k \right\rceil - 2^n x\right| < 1$$

Proof

$$\begin{aligned}\left|\left\lfloor\frac{\mathbf{x}(n+k)}{2^k}\right\rceil - 2^n x\right| &\leq \frac{1}{2} + \left|\frac{\mathbf{x}(n+k)}{2^k} - 2^n x\right| \\ &= \frac{1}{2} + \frac{1}{2^k}\left|\mathbf{x}(n+k) - 2^{n+k}x\right| \\ &< \frac{1}{2} + \frac{1}{2^k} \\ &\leq 1\end{aligned}$$

Theorem A.0.5 (msd existence)

$$\forall x \neq 0.\ \exists n.\ |\mathbf{x}(n)| > 1.$$

Proof From $\lfloor x \rfloor \leq x \leq \lfloor x \rfloor + 1$ we have

$$2^{\lfloor \log_2 |x| \rfloor} < |x| < 2^{\lfloor \log_2 |x| \rfloor + 1}. \tag{A.1}$$

Also, by definition, we have

$$\frac{|\mathbf{x}(n)| - 1}{2^n} < |x| < \frac{|\mathbf{x}(n)| + 1}{2^n} \tag{A.2}$$

From A.2 and letting $n = -\lfloor \log_2 |x| \rfloor$ we get

$$\frac{|\mathbf{x}(-\lfloor \log_2 |x| \rfloor)| - 1}{2^{-\lfloor \log_2 |x| \rfloor}} < |x| < \frac{|\mathbf{x}(-\lfloor \log_2 |x| \rfloor)| + 1}{2^{-\lfloor \log_2 |x| \rfloor}}. \tag{A.3}$$

Combining A.1 and A.3 we get

$$\frac{|\mathbf{x}(-\lfloor \log_2 |x| \rfloor)| - 1}{2^{-\lfloor \log_2 |x| \rfloor}} < 2^{\lfloor \log_2 |x| \rfloor + 1} \quad \text{and} \quad 2^{\lfloor \log_2 |x| \rfloor} < \frac{|\mathbf{x}(-\lfloor \log_2 |x| \rfloor)| + 1}{2^{-\lfloor \log_2 |x| \rfloor}}, \tag{A.4}$$

or equivalently,

$$|\mathbf{x}(-\lfloor \log_2 |x| \rfloor)| - 1 < 2 \quad \text{and} \quad 1 < |\mathbf{x}(-\lfloor \log_2 |x| \rfloor)| + 1. \tag{A.5}$$

But since $\mathbf{x}(-\lfloor \log_2 |x| \rfloor) \in \mathbb{Z}$, A.5 is same as

$$|\mathbf{x}(-\lfloor \log_2 |x| \rfloor)| \leq 2 \quad \text{and} \quad 1 \leq |\mathbf{x}(-\lfloor \log_2 |x| \rfloor)| \tag{A.6}$$

or,

$$1 \leq |\mathbf{x}(-\lfloor \log_2 |x| \rfloor)| \leq 2. \tag{A.7}$$

Now assuming $|\mathbf{x}(-\lfloor \log_2 |x| \rfloor)| = 1$ (the smallest case) we show that

$$1 < |\mathbf{x}(-\lfloor \log_2 |x| \rfloor + 1)|. \tag{A.8}$$

From A.2 and letting $n = -\lfloor \log_2 |x| \rfloor + 1$ we have

$$\frac{|\mathbf{x}(-\lfloor \log_2 |x| \rfloor + 1)| - 1}{2^{-\lfloor \log_2 |x| \rfloor + 1}} < |x| < \frac{|\mathbf{x}(-\lfloor \log_2 |x| \rfloor + 1)| + 1}{2^{-\lfloor \log_2 |x| \rfloor + 1}}. \tag{A.9}$$

Combining A.9 with A.1 we get

$$\frac{|\mathbf{x}(-\lfloor \log_2 |x| \rfloor + 1)| - 1}{2^{-\lfloor \log_2 |x| \rfloor + 1}} < 2^{\lfloor \log_2 |x| \rfloor + 1} \tag{A.10}$$

and

$$2^{\lfloor \log_2 |x| \rfloor} < \frac{|\mathbf{x}(-\lfloor \log_2 |x| \rfloor + 1)| + 1}{2^{-\lfloor \log_2 |x| \rfloor + 1}} \tag{A.11}$$

or equivalently,

$$|\mathbf{x}(-\lfloor \log_2 |x| \rfloor + 1)| - 1 < 2^2 \quad \text{and} \quad 2 < |\mathbf{x}(-\lfloor \log_2 |x| \rfloor + 1)| + 1. \tag{A.12}$$

A.12 is equivalent to

$$2 \leq |\mathbf{x}(-\lfloor \log_2 |x| \rfloor + 1)| \leq 4 \tag{A.13}$$

Since for a positive integer k, $k \geq 2 \Rightarrow k > 1$, we have shown A.8. Letting $n = -\lfloor \log_2 |x| \rfloor + 1$ shows the theorem.

Theorem A.0.6 $\vdash \forall n.\ n < -\lfloor \log_2 |\mathbf{x}| \rfloor \Rightarrow |\mathbf{x}(n)| \leq 1$
Proof From A.1 and A.2 we have

$$|\mathbf{x}(n)| - 1 \quad < \quad 2^{n + \lfloor \log_2 |x| \rfloor + 1}. \tag{A.14}$$

From the assumption we have

$$n + \lfloor \log_2 |x| \rfloor + 1 \quad \leq \quad 0 \tag{A.15}$$

From A.14 and A.15 we get $|\mathbf{x}(n)| < 2$. Since $\mathbf{x}(n) \in Z$ this means $|\mathbf{x}(n)| \leq 1$.

Theorem A.0.7 $\mathtt{msd}(\mathbf{x}) = -\lfloor \log_2 |\mathbf{x}| \rfloor \Rightarrow 1 < |\mathbf{x}(\mathtt{msd})| \leq 2$
Proof By Substituition into A.14.

Theorem A.0.8 $\mathtt{msd}(\mathbf{x}) = -\lfloor \log_2 |\mathbf{x}| \rfloor + 1 \Rightarrow 2 \leq |\mathbf{x}(\mathtt{msd})| \leq 4$
Proof $|\mathbf{x}(-\lfloor \log_2 |x| \rfloor)| = 1$. Thus $|\mathbf{x}(-\lfloor \log_2 |x| \rfloor)| + 1 = 2$. Also $|x| < 2 \times 2^{\lfloor \log_2 |x| \rfloor}$ and $|\mathbf{x}(\mathrm{msd})| - 1 < 2 \times 2$. Hence $2 \leq |\mathbf{x}(\mathtt{msd})| \leq 2 \times 2$.

Theorem A.0.9 $\forall n \geq \mathtt{msd}(\mathbf{x}).\ 2^{n - \mathtt{msd}(\mathbf{x})} \leq |\mathbf{x}(n)| \leq 2^{n - \mathtt{msd}(\mathbf{x})} \times 5$
Proof We have

$$(|\mathbf{x}(\mathtt{msd})| - 1)2^{n - \mathtt{msd}(\mathbf{x})}2^{-n} < |x| < (|\mathbf{x}(\mathtt{msd})| - 1)2^{n - \mathtt{msd}(\mathbf{x})}2^{-n}$$

thus

$$(|\mathbf{x}(\mathtt{msd})| - 1)2^{n - \mathtt{msd}(\mathbf{x})} \leq 2^n x \leq (|\mathbf{x}(\mathtt{msd})| - 1)2^{n - \mathtt{msd}(\mathbf{x})}$$

and

$$(|\mathbf{x}(\mathbf{msd})| - 1)2^{n-\mathbf{msd}(\mathbf{x})} \leq |\mathbf{x}(n)| \leq (|\mathbf{x}(\mathbf{msd})| - 1)2^{n-\mathbf{msd}(\mathbf{x})}$$

but $2 \leq |\mathbf{x}(\mathbf{msd}(\mathbf{x}))| \leq 4$, thus

$$2^{n-\mathbf{msd}(\mathbf{x})} \leq |\mathbf{x}(n)| \leq 2^{n-\mathbf{msd}(\mathbf{x})}(2 \times 2 + 1).$$

Theorem A.0.10 $1 \leq \left|\lfloor \frac{\mathbf{x}(n)}{2^{n-\mathbf{msd}(\mathbf{x})}} \rfloor\right| \leq 5$

Proof We can assume that $|\mathbf{x}(n)| \geq 2 \times 2 \times 2^{n-\mathbf{msd}(\mathbf{x}+1)} + 1$. But this is impossible if $n = \mathbf{msd}(\mathbf{x})$ so we can consider directly that $n - \mathbf{msd}(\mathbf{x}) \geq 1$. Thus we have $2^{\mathbf{msd}(\mathbf{x})-1}|x| > (|\mathbf{x}(n)| - 1)2^{-(n-\mathbf{msd}(\mathbf{x})+1)} \geq 2$, thus $|\mathbf{x}(\mathbf{msd} - 1)| \geq 2$. This is impossible by the minimality of $\mathrm{msd}(\mathbf{x})$. Consequently

$$\frac{|\mathbf{x}(n)|}{2^{n-\mathbf{msd}(\mathbf{x})}} \leq 2 \times 2$$

and

$$\lfloor \frac{|\mathbf{x}(n)|}{2^{n-\mathbf{msd}(\mathbf{x})}} \rfloor \leq 2 \times 2$$

But we may have $2^{n-\mathbf{msd}(\mathbf{x})} \leq |\mathbf{x}(n)| < 2^{n-\mathbf{msd}(\mathbf{x})+1}$ for $n > \mathbf{msd}(\mathbf{x}) + 1$

Definition A.0.11 (x+y) $\mathbf{x} + \mathbf{y} := n \mapsto \lfloor \frac{\mathbf{x}(n+2)+\mathbf{y}(n+2)}{4} \rceil$

Proof

$$\begin{aligned}
|(\mathbf{x}+\mathbf{y})(n) - 2^n(x+y)| &= \left|\left\lfloor \frac{\mathbf{x}(n+2)+\mathbf{y}(n+2)}{4} \right\rceil - 2^n(x+y)\right| \\
&\leq \frac{1}{2} + \left|\frac{\mathbf{x}(n+2)+\mathbf{y}(n+2)}{4} - 2^n(x+y)\right| \\
&= \frac{1}{2} + \frac{1}{4}\left|(\mathbf{x}(n+2)+\mathbf{y}(n+2)) - 2^{n+2}(x+y)\right| \\
&\leq \frac{1}{2} + \frac{1}{4}\left|\mathbf{x}(n+2) - 2^{n+2}x\right| + \frac{1}{4}\left|\mathbf{y}(n+2) - 2^{n+2}y\right| \\
&< \frac{1}{2} + \frac{1}{4} + \frac{1}{4} \\
&= 1
\end{aligned}$$

Definition A.0.12 ($k\mathbf{x}$ for some integer k) $k\mathbf{x} := n \mapsto \left\lfloor \frac{k\mathbf{x}(n+p+1)}{2^{p+1}} \right\rceil$

Proof Choose p so that $|k|/2^p \leq 1$.

$$\begin{aligned}
|k\mathbf{x}(n) - 2^n(kx)| &= \left|\left\lfloor \frac{k\mathbf{x}(n+p+1)}{2^{p+1}} \right\rfloor - 2^n(kx)\right| \\
&\leq \frac{1}{2} + \left|\frac{k\mathbf{x}(n+p+1)}{2^{p+1}} - 2^n(kx)\right| \\
&= \frac{1}{2} + \frac{|k|}{2^{p+1}}\left|\mathbf{x}(n+p+1) - 2^{n+p+1}x\right| \\
&< \frac{1}{2} + \frac{|k|}{2^{p+1}} \\
&\leq \frac{1}{2} + \frac{1}{2}\frac{|k|}{2^p} \\
&\leq \frac{1}{2} + \frac{1}{2} \quad (\text{since } |k|/2^m \leq 1) \\
&= 1
\end{aligned}$$

Definition A.0.13 ($\mathbf{x}/k$ **for some integer** $k \neq 0$) $\mathbf{x}/k := n \mapsto \left\lfloor \frac{\mathbf{x}(n)}{k} \right\rfloor$

Proof We can assume k is positive. If $k = 1$, then $|(\mathbf{x}/1)(n) - 2^n(x/1) = |\lfloor \mathbf{x}(n)/1 \rfloor - 2^n x| = |\mathbf{x}(n) - 2^n x| < 1$. Now assume $k \geq 2$.

$$\begin{aligned}
|(\mathbf{x}/k)(n) - 2^n(x/k)| &= \left|\left\lfloor \frac{\mathbf{x}(n)}{k} \right\rfloor - 2^n(x/k)\right| \\
&\leq \frac{1}{2} + \left|\frac{\mathbf{x}(n)}{k} - 2^n(x/k)\right| \\
&= \frac{1}{2} + \frac{1}{k}\left|\mathbf{x}(n) - 2^n x\right| \\
&\leq \frac{1}{2} + \frac{1}{k} \\
&\leq \frac{1}{2} + \frac{1}{2} \\
&= 1
\end{aligned}$$

Definition A.0.14 ($\sum_{i=0}^{m-1} \mathbf{x}_i$) $\sum_{i=0}^{m-1} \mathbf{x}_i := n \mapsto \left\lfloor \frac{\sum_{i=0}^{m-1} \mathbf{x}_i(n+p+1)}{2^{p+1}} \right\rfloor$ *where* $m \leq 2^p$

Proof

$$\begin{aligned}
\left|\left\lfloor \frac{\sum_{i=0}^{m-1} \mathbf{x}_i(n+p+1)}{2^{p+1}} \right\rfloor - 2^n \sum_{i=0}^{m-1} x_i\right| &\leq \frac{1}{2} + \frac{1}{2^{p+1}}\left|\sum_{i=0}^{m-1} \mathbf{x}_i(n+p+1) - 2^{n+p+1}\sum_{i=0}^{m-1} x_i\right| \\
&\leq \frac{1}{2} + \frac{2^p}{2^{p+1}} \\
&< 1
\end{aligned}$$

For proofs of multiplication and division see the mentioned references above.

Appendix B

log, sin **and** arctan **in** $\mathbb{R}_{ldr}$

Here we prove the algorithms for `ln(x)`, `sin(x)` and `arctan(x)` in $\mathbb{R}_{ldr}$.

B.1 $\texttt{ln}\frac{(1+\texttt{x})}{(1-\texttt{x})}$ for $|\texttt{x}| < \frac{1}{2}$

An upper bound for $\texttt{ln}\frac{(1+\texttt{x})}{(1-\texttt{x})}$ for $|\texttt{x}| < \frac{1}{2}$ can be derived from the Lagrange version of Taylor's theorem as we did for `exp(x)` but we will take more explicit approach using the Taylor series for $\ln\frac{(1+x)}{(1-x)}$. We start with the Taylor series for $\ln(1+x)$. For $x > -1$,

$$\begin{aligned}
\ln 1 + x &= \int_{t=0}^{x} \frac{dt}{1+t} \\
&= \sum_{i=0}^{m-1} (-1)^i \frac{x^{i+1}}{i+1} + (-1)^{m-1} \int_{t=0}^{x} \frac{t^{m-1}}{1+t} dt \\
&= (x - \frac{x^2}{2} + \frac{x^3}{3} - \frac{x^4}{4} + \frac{x^5}{5}) - \int_{t=0}^{x} \frac{t^{m-1}}{1+t} dt
\end{aligned}$$

and by substituting $-x$ for x

$$\begin{aligned}
\ln(1-x) &= \int_{t=0}^{x} \frac{dt}{1+t} \\
&= \sum_{i=0}^{m-1} (-1)^{i+1} \frac{x^{i+1}}{i+1} + (-1)^m \int_{t=0}^{x} \frac{t^{m-1}}{1+t} dt \\
&= (-x - \frac{x^2}{2} - \frac{x^3}{3} - \frac{x^4}{4} - \frac{x^5}{5}) - \int_{t=0}^{x} \frac{t^{m-1}}{1+t} dt
\end{aligned}$$

Hence (for $|x| \leq 1/2$)

$$\begin{aligned}
\ln \frac{(1+x)}{(1-x)} &= \ln 1 + x - \ln 1 - x \\
&= 2 \sum_{i=0}^{m-1} \frac{x^{2i+1}}{2i+1} + 2(-1)^{m-1} \int_{t=0}^{x} t^{m-1} dt
\end{aligned}$$

For $|x| \leq 1/2$,

$$\begin{aligned} 2\left|(-1)^{m-1}\int_{t=0}^{x}\frac{t^{m-1}}{1+t}dt\right| &= 2\left|\int_{t=0}^{x}\frac{t^{m-1}}{1+t}dt\right| \\ &\leq 2\left|\int_{t=0}^{x}t^{m-1}dt\right| \\ &\leq \frac{1}{2^{m-1}m}. \end{aligned}$$

Hence the truncation error satisfies the inequality

$$\left|\int_{t=0}^{x}t^{m-1}dt\right| \leq \frac{1}{2^{m-2}m}$$

Next we derive the relation between $\mathtt{k}_{i+1}(n)$ and $\mathtt{k}_i(n)$ for the two consecutive terms of the Taylor series for $\mathtt{ln}\frac{(1+x)}{(1-x)}$. Assume that

$$\left|\mathtt{t}_i(n) - 2^n\frac{x^{2i+1}}{2i+1}\right| \leq \mathtt{k}_i(n).$$

and then calculate $\mathtt{t}_{i+1}$ by

$$\mathtt{t}_{i+1} := n \mapsto ((2i+1)\mathtt{x}^2(n)\mathtt{t}_i(n))\ \mathtt{ndiv}\ (2^n(2i+3)).$$

Then we get a formula as below

$$\mathtt{k}_{i+1}(n) = \frac{2(2i+1)}{2i+3}\mathtt{k}_i(n) + \frac{1}{(2i+3)} + \frac{1}{2}$$

since

$$\begin{aligned} \left|\mathtt{t}_{i+1}(n) - 2^n\frac{x^{2i+3}}{(2i+3)}\right| &\leq \left|\frac{(2i+1)\mathtt{x}^2(n)}{2^n(2i+3)}\mathtt{t}_i(n) - \frac{(2i+1)\mathtt{x}^2(n)}{2^n(2i+3)}\frac{2^n x^{2i+1}}{2+1}\right| + \\ &\qquad \frac{1}{2} + \left|\frac{(2i+1)\mathtt{x}^2(n)}{2^n(2i+3)}\frac{2^n x^{2i+1}}{2+1} - \frac{2^n x^{2i+3}}{2i+3}\right| \\ &= \frac{1}{2} + \left|\frac{(2i+1)\mathtt{x}^2(n)}{2^n(2i+3)}\right|\left|\mathtt{t}_i(n) - \frac{2^n x^{2i+1}}{2i+1}\right| + \frac{x^{2i+2}}{2i+3}\left|\mathtt{x}(n) - 2^n x\right| \\ &\leq \frac{1}{2} + \left|\frac{(2i+1)\mathtt{x}^2(n)}{2^n(2i+3)}\right|\mathtt{k}_i + \frac{x^{2i+2}}{2i+3} \\ &= \frac{1}{2} + \frac{(2i+1)|\mathtt{x}(n)|}{2i+3}\mathtt{k}_i\left(\left|\frac{\mathtt{x}(n)}{2^n} - |x|\right| + |x|\right) + \frac{x^{2i+2}}{2i+3} \\ &\leq \frac{1}{2} + \frac{2(2i+1)}{2i+3}\mathtt{k}_i + \frac{1}{2i+3}. \end{aligned}$$

Hence by induction we can conclude that for all i, $\mathtt{k}_i(n) \leq 2$.

Proof

$$\mathtt{k}_{i+1}(n) = \frac{2(2i+1)}{(2i+3)}\mathtt{k}_i(n) + \frac{1}{2i+3} + \frac{1}{2}$$

$$
\begin{aligned}
&= \frac{2i+1}{2i+3}\mathtt{k}_i(n) + \frac{2i+5}{2(2i+3)} \\
&\leq 2
\end{aligned}
$$

since $\frac{2i+1}{2i+3} < 1$, $\mathtt{k}_i(n) < 1$ (by assumption) and $\frac{2i+5}{2(2i+3)} < 1$.
Thus the total total summation error of m-terms is always $< 2m$. Now we can write an algorithm for $\ln \frac{(1+\mathtt{x})}{(1-\mathtt{x})}$ for $|\mathtt{x}| \leq 1/2$. First find an m and e such that

$$
\frac{1}{2^{m-2}m} \quad < \quad \frac{1}{2^{n+2}}
$$

and

$$
2m \quad \leq \quad 2^e.
$$

Then evaluate $\mathtt{x}$ to the precision $n+e+2$. Summarizing all this below.

Theorem B.1.1 *Let*

$$
\mathtt{ln}\frac{1+\mathtt{x}}{1-\mathtt{x}} \quad = \quad n \mapsto \sum_{i=0}^{m-1} \mathtt{t}_i(n) \; \mathit{ndiv}\; 2^{e+2}
$$

where

$$
\mathtt{t}_i(n) \quad = \quad \begin{cases} 2^{n+e+2}\mathtt{x}(n+e+2) & \mathit{if}\; i = 0 \\ ((2i+1)\mathtt{x}^2(n)\mathtt{t}_{i-1}(n))\;\mathtt{ndiv}\; 2^n(2i+3) & \mathit{otherwise.} \end{cases}
$$

Then

$$
\left|\mathtt{ln}\frac{1+\mathtt{x}}{1-\mathtt{x}}(n) - 2^n \ln\frac{(1+x)}{(1-x)}\right| \quad < \quad 1.
$$

B.2 sin(x) for $0 \leq \mathtt{x} \leq \pi/4$

Applying the Lagrange version of Taylor's theorem for $\sin x$ with $a = 0$ and $\xi = \pi/4$ gives

$$
\sin(x) \quad = \quad \sum_{i=0}^{m-1} (-1)^i \frac{x^{2i+1}}{(2i+1)!} + (-1)^i \frac{x^m}{m!} \sin^{(m)}(\pi/4).
$$

Since $\sin(\pi/4) = \cos(\pi/4) = \sqrt{2}/2 < 1$ and $|x| \leq \pi/4$ the truncation error is

$$
\left|(-1)^i \frac{x^m}{m!} \sin^{(m)}(\pi/4)\right| \quad \leq \quad \frac{3}{2m!}
$$

Next, for the modelling error, let $\mathtt{k}_i(n)$ be the guaranteed error bound for the i-th term, i.e.,

$$\left|\mathtt{t}_i(n) - 2^n(-1)^i\frac{x^{2i+1}}{(2i+1)!}\right| \;\leq\; \mathtt{k}_i(n)$$

and calculate the next term $\mathtt{t}_{i+1}(n)$ using the formula

$$\mathtt{t}_{i+1} \;:=\; n \mapsto (\mathtt{x}^2(n)\mathtt{t}_i(n))\ \mathtt{ndiv}\ (2^n(2i+2)(2i+3))$$

Then we get a formula as below

$$\mathtt{k}_{i+1}(n) \;=\; \frac{2}{(2i+2)(2i+3)}\mathtt{k}_i(n) + \frac{1}{(2i+3)!} + \frac{1}{2}$$

since

$$\begin{aligned}
\left|\mathtt{t}_{i+1}(n) - 2^n\frac{x^{2i+3}}{(2i+3)!}\right| &\leq \left|\frac{\mathtt{x}^2(n)}{2^n(2i+2)(2i+3)}\mathtt{t}_i(n) - \frac{\mathtt{x}^2(n)}{2^n(2i+2)(2i+3)}\frac{2^n x^{2i+1}}{(2i+1)!}\right| \\
&\quad + \left|\frac{\mathtt{x}^2(n)}{2^n(2i+2)(2i+3)}\frac{2^n x^{2i+1}}{(2i+1)!} - \frac{2^n x^{2i+3}}{(2i+3)!}\right| + \frac{1}{2} \\
&= \left|\frac{\mathtt{x}^2(n)}{2^n(2i+2)(2i+3)}\right|\left|\mathtt{t}_i(n) - \frac{2^n x^{2i+1}}{(2i+1)!}\right| + \frac{x^{2i+2}}{(2i+3)!}\left|\mathtt{x}(n) - 2^n x\right| + \frac{1}{2} \\
&\leq \left|\frac{\mathtt{x}^2(n)}{2^n(2i+2)(2i+3)}\right|\mathtt{k}_i(n) + \frac{x^{2i+2}}{(2i+3)!} + \frac{1}{2} \\
&= \frac{|\mathtt{x}(n)|}{(2i+2)(2i+3)}k_i(n)\left(\left|\frac{\mathtt{x}(n)}{2^n} - |x|\right| + |x|\right) + \frac{x^{2i+2}}{(2i+3)!} + \frac{1}{2} \\
&\leq \frac{2}{(2i+2)(2i+3)}\mathtt{k}_i(n) + \frac{1}{2i+3} + \frac{1}{2}.
\end{aligned}$$

Hence by induction we can conclude that $\mathtt{k}_i(n) \leq 2$ for all i since

Proof

$$\begin{aligned}
\mathtt{k}_{i+1}(n) &= \frac{2}{(2i+2)(2i+3)}\mathtt{k}_i(n) + \frac{1}{2i+3} + \frac{1}{2} \\
&= \frac{1}{(2i+2)(2i+3)}\mathtt{k}_i(n) + \frac{2i+5}{2(2i+3)} \\
&\leq 2
\end{aligned}$$

since $1/((2i+2)(2i+3)) < 1$, $\mathtt{k}_i(n) < 1$ (by assumption) and $(2i+5)/2(2i+3) < 1$. Thus the total total summation error of m-terms is always $< 2m$. Now we can write an algorithm for `sin(x)` for $0 \leq \mathtt{x} \leq \pi/4$. First find an m and e such that

$$\frac{3}{2m!} < \frac{1}{2^{n+2}}2m \;\leq\; 2^e$$

and

$$2m \;\leq\; 2^e.$$

Then evaluate $\mathtt{x}$ to the precision $n+e+2$. In summary

Theorem B.2.1 *Let*

$$\mathtt{sin(x)} \;:=\; n \mapsto \sum_{i=0}^{m-1} \mathtt{t}_i(n) \;\mathtt{ndiv}\; 2^{e+2}$$

where

$$\mathtt{t}_i \;:=\; n \mapsto \begin{cases} 2^{n+e+2}\mathtt{x}(n+e+2) & \text{if } i = 0 \\ (\mathtt{x}^2(n)\mathtt{t}_{i-1}(n)) \;\mathtt{ndiv}\; 2^n(2i+2)(2i+3) & \text{otherwise.} \end{cases}$$

Then

$$|\mathtt{sin(x)}(n) - 2^n \sin x| \;<\; 1.$$

B.3 arctan(x) for $|\mathtt{x}| \leq 1$

Again from the Lagrange version of Taylor's theorem for $\arctan(x)$ with $a = 0$ and $\xi = 1$ we get

$$\arctan(x) = \sum_{i=0}^{m-1} (-1)^i \frac{x^{2i+1}}{2i+1} + (-1)^i \frac{x^m}{m!} \arctan^{(m)}(1).$$

Since $\arctan^{(m)}(1) < 1$ for all m and $|x| \leq 1$ for the truncation error we have

$$\left|(-1)^i \frac{x^m}{m!} \arctan^{(m)}(1)\right| \;\leq\; \frac{1}{m!}$$

Next, let $\mathtt{k}_i(n)$ be the guaranteed error bound for the i-th term, i.e.,

$$\left|\mathtt{t}_i(n) - 2^n(-1)^i \frac{x^{2i+1}}{(2i+1)}\right| \;\leq\; \mathtt{k}_i(n)$$

and calculate the next term $\mathtt{t}_{i+1}(n)$ using the formula

$$\mathtt{t}_{i+1} \;:=\; n \mapsto ((2i+1)\mathtt{x}^2(n)\mathtt{t}_i(n)) \;\mathtt{ndiv}\; (2^n(2i+3))$$

Then by similar error analysis we get the formula

$$\mathtt{k}_{i+1}(n) \;=\; \frac{2(2i+1)}{2i+3}\mathtt{k}_i(n) + \frac{1}{2i+3} + \frac{1}{2}$$

sicne

$$\begin{aligned}\left|\mathtt{t}_{i+1}(n) - 2^n \frac{x^{2i+3}}{2i+3}\right| \;\leq\; & \left|\frac{(2i+1)\mathtt{x}^2(n)}{2^n(2i+3)}\mathtt{t}_i(n) - \frac{(2i+1)\mathtt{x}^2(n)}{2^n(2i+3)}\frac{2^n x^{2i+1}}{2i+1}\right| \\ & + \left|\frac{(2i+1)\mathtt{x}^2(n)}{2^n(2i+3)}\frac{2^n x^{2i+1}}{2i+1} - \frac{2^n x^{2i+3}}{2i+3}\right| + \frac{1}{2}\end{aligned}$$

$$
\begin{aligned}
&= \left|\frac{(2i+1)\mathtt{x}^2(n)}{2^n(2i+3)}\right|\left|\mathtt{t}_i(n) - \frac{2^n x^{2i+1}}{2i+1}\right| + \frac{x^{2i+2}}{2i+3}\,|\mathtt{x}(n) - 2^n x| + \frac{1}{2} \\
&\leq \left|\frac{(2i+1)\mathtt{x}^2(n)}{2^n(2i+3)}\right|\mathtt{k}_i(n) + \frac{x^{2i+2}}{2i+3} + \frac{1}{2} \\
&= \frac{(2i+1)|\mathtt{x}(n)|}{(2i+3)}\mathtt{k}_i(n)\left(\left|\frac{\mathtt{x}(n)}{2^n} - |x|\right| + |x|\right) + \frac{x^{2i+2}}{2i+3} + \frac{1}{2} \\
&\leq \frac{2(2i+1)}{2i+3}\mathtt{k}_i + \frac{1}{2i+3} + \frac{1}{2}.
\end{aligned}
$$

Hence by induction we can conclude that $\mathtt{k}_i(n) \leq 2$ for all i since

Proof

$$
\begin{aligned}
\mathtt{k}_{i+1}(n) &= \frac{2(2i+1)}{2i+3}\mathtt{k}_i(n) + \frac{1}{2i+3} + \frac{1}{2} \\
&= \frac{2i+1}{2i+3}\mathtt{k}_i + \frac{2i+5}{2(2i+3)} \\
&\leq 2
\end{aligned}
$$

since $(2i+1)/(2i+3) < 1, (2i+5)/2(2i+3) < 1$ for all i and $\mathtt{k}_i < 1$ (by assumption). Thus the total total summation error of m-terms is always $< 2m$. Now we can write an algorithm for `arctan(x)` for $|\mathtt{x}| \leq 1$. We find an m and e such that

$$
\frac{1}{m!} < \frac{1}{2^{n+2}}.
$$

and

$$
2m \leq 2^e.
$$

Then evaluate $\mathtt{x}$ to the precision $n+e+2$. In summary

Theorem B.3.1 *Let*

$$
\mathtt{arctan(x)} := n \mapsto \sum_{i=0}^{m-1} \mathtt{t}_i(n) \ \mathtt{ndiv}\ 2^{e+2}
$$

where

$$
\mathtt{t}_i := n \mapsto \begin{cases} 2^{n+e+2}\mathtt{x}(n+e+2) & \text{if } i = 0 \\ ((2i+1)\mathtt{x}^2(n)\mathtt{t}_{i-1}(n)) \ \mathtt{ndiv}\ 2^n(2i+3) & \text{otherwise.} \end{cases}
$$

Then

$$
|\mathtt{arctan(x)}(n) - 2^n \arctan x| < 1.
$$

Bibliography

[1] How does one program in the axiom system, 1992. NAG Technical Report.

[2] Z. Adamowicz and P. Zbierski. *Logic of Mathematics: a modern course of classical logic.* John Wiley and Sons, 1997.

[3] D. H. Bailey, J. M. Borwein, P. B. Borwein, and S. Plouffe. The quest for pi. http://www.cecm.sfu.ca/ pborwein/PAPERS/P130.ps, June 1996.

[4] Bridges D. Bishop, E. *Constructive Analysis.* Springer Verlag, 1985.

[5] H-J. Boehm. Constructive real interpretation of numerical programs. In *Proceedings of the SIGPLAN '87 Symposium on Interpreters and Interpretive Techniques*, pages 214–221. ACM Press, June 1987.

[6] H-J. Boehm and R. Cartwright. Exact real arithmetic, formulating real numbers as functions. In David A. Turner, editor, *Research Topics in Functional Programming*, pages 43–64. Addison-Wesley, 1990.

[7] H-J. Boehm, R. Cartwright, M. J. O'Donnel, and M. Riggle. Exact real arithmetic, a case study in higher order programming. In *Proceedings of the ACM conference on Lisp and functional programming.*

[8] R. P. Brent. Fast multiple-precision evaluation of elementary functions. *Journal of the ACM*, 23:242–251, 1976.

[9] K. Mehlhorn C. Burnikel, K. Fleischer and S. Schirra. Exact efficient computational geometry made easy. In *Proceedings of the 15th Annual Symposium on Computational Geometry (SCG'99)*, pages 341–350, 1999.

[10] K. Ciesielski. *Set Theory for the Working Mathematician.* LMS Student Texts 39. CUP, 1997.

[11] H. Cohen. *A Course in Computational Algebraic Number Theory.* Springer-Verlag, 1993.

[12] J. H. Davenport. Equality in computer algebra and beyond. To appear in Proc. of Calculemus 2001.

[13] J. H. Davenport. Computer algebra for cylindrical algebraic decomposition, 1988. Available at `http://www.bath.ac.uk/~masjhd/TRITA.dvi`.

[14] J. H. Davenport, Y. Siret, and E. Tournier. *Computer Algebra.* Academic Press, 2nd edition, 1993.

[15] R. W. Gosper. Item 101b: Continued fraction arithmetic. *HAKMEM, MIT Artificial Intelligence Memo 239*, pages 39–44, February 1972.

[16] E. Hairer and G. Wanner. *Analysis by Its History.* Undergraduate Texts in Mathematics. Springer, 1995.

[17] M. Hall Jr. On the sum and product of continued fractions. *Annals of Math.*, 48:966–993, 1947.

[18] J. R. Harrison. Introduction to functional programming. Available at `http://www.cl.cam.ac.uk/~jrh`.

[19] J. R. Harrison. *Theorem Proving with the Real Numbers.* Springer-Verlag, 1998.

[20] R. Heckmann. The appearance of big integers in exact real arithmetic based on linear fractional transformations. In *Foundations of Software Science and Computation Structures*, LNCS 1378, pages 172–188. Springer, 1998.

[21] R. Heckmann. Big integers and complexity issues in exact real arithmetic. In *Electronic Notes in Theoretical Computer Science*, volume 13, 1998. Available at `http://www.elsevier.nl/locate/entcs/volume13.html`.

[22] R. Heckmann. Contractivity of linear fractional transformatiosn. In *Third Real Numbers and Computers Conference (RNC3)*, pages 45–59, 1998. An updated version will appear in TCS.

[23] R. Heckmann. Translation of taylor series into lft expansions. 1999. submitted to Proceedings of Dagstuhl Seminar " Symbolic Algebraic Mehtod and Verification Methods.

[24] R. Heckmann. How many argument digits are needed to produce n result digits? In *Electronic Notes in Theoretical Computer Science*, volume 24, 2000. Available at `http://www.elsevier.nl/locate/entcs/volume24.html`.

[25] R. D. Jenks and R. S. Sutor. *Axiom: The Scientific Computation System.* Springer-Verlag, 1992.

[26] Musser D. Saunders B. Kaltofen, E. A generalized class of polynomials that are hard to factor. *SIAM Journal of Computing*, 12(3):473–483, 1983.

[27] A Khinchin. *Continued Fractions.* University of Chicago Press, 1964.

[28] Ker-I Ko. *Complexity Theory of Real Functions.* Birkhäuser, 1991.

[29] S. Landau. How to tangle with a nested radical. *The Mathematical Intelligencer*, 16(2):49–55.

[30] Lenstra Jr. H. W. Lenstra, A. K. and L. Lovász. Factoring polynomials with rational coefficients. *Math. Ann.*, 261:515–534, 1982.

[31] D. R. Lester. Vuillemin's exact real arithmetic. In *Functional Programmin: Proceedings of the 1991 Workshop, Glasgow 1991*, pages 225–238. Springer Verlag, 1992.

[32] R. Loos. Computing in algebraic extensions. *Computing*, 4 (suppl.):173–187.

[33] V. Ménissier-Morain. *Arithméique exacte, conception, algorithmique et performances d'une implmentation informatique en pré cision arbitraire.* PhD thesis, Université Paris 7, 1994.

[34] V. Ménissier-Morain. Arbitrary precision real arithmetic: design and algorithms. *Journal of Symbolic Computation*, 11:1–100, 1996.

[35] B. Mishra. *Algorithmic Algebra.* Texts and monographs in computer science. Springer-Verlag, 1993.

[36] J. Myhill. What is a real number? *American Mathematical Monthly*, pages 748–754, 1972.

[37] Edalat A. Potts, P. J. and M. H. Escardó. Semantics of exact computer arithmetic. In *12th Annual IEEE Symp. on Logic in Computer Science*, pages 248–257. IEEE Computer Society Press, 1997.

[38] P. J. Potts. *Exact Real Arithmetic using Möbius Transformations.* PhD thesis, Imperial College, 1998.

[39] M. B. Pour-El and I. J. Richards. *Computability in Analysis and Physics.* Springer-Verlag, 1989.

[40] H. G. Rice. Recursive real numbers. *Proceedings of the American Mathematical Society*, 5:784–791, 1954.

[41] D. Richardson. How to recognize zero. *Journal of Symbolic Computation*, 24:1–19, 1997.

[42] R. Rioboo. Real algebraic closure of an ordered field, implementation in axiom. In *Proc. ISSAC'92*, pages 206–215. ISSAC, ACM Press, 1992.

[43] E. Specker. Nicht konstruktiv beweisbare sätze der analysis. *J. Symbolic Logic*, 14:145–158, 1949.

[44] A. W. Strzeboński. Computing in the field of complex algebraic numbers. *Journal of Symbolic Computation*, 24:647–656, 1997.

[45] P. Sünderhauf. Incremental addition in exact real arithmetic. Technical report, Imperial College, 1997.

[46] I. Vardi. Code and pseudo code. *Matehmatica* Journal, 1996. Issue 2, pp. 66-71.

[47] J. E. Vuillemin. Exact real computer arithmetic with continued fractions. *IEEE Trans. on Comput.*, 39(8):1087–1105, August 1990.

[48] A. J. Wilkie. Model completeness results for expansions of the ordered field of real numbers by restricted pfaffian functions and the exponential function. *Journal of the AMS*, 9:1051–1094, 1996.

[49] W-T Wu. *Mechanical theorem proving in geometries: basic principles.* Springer, 1994.

[50] C. Yap. *Fundamental Problems in Algorithmic Algebra.* Oxford University Press, 1999.